



University of
Pittsburgh

Algorithms and Data Structures 2

CS 1501



Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
 - Homework 4: this Friday @ 11:59 pm
 - Lab 3: next Monday @ 11:59 pm
 - Assignment 1: Monday Oct 10th @ 11:59 pm
- **Live support session** for Assignment 1
 - Over Zoom this Friday @ 5:00 pm
- **Student Support Hours** of the teaching team are posted on the Syllabus page

Previous lecture

- Digital Searching Problem
 - Searching when keys are represented as a sequence of digits (e.g., bits) or alphabetic characters
 - Digital Search Trees
 - Radix Search Tries

This Lecture

- R-way Radix Search Tries
- De La Briandais (DLB) Tries

Adding to Radix Search Trie (RST)

- Input: key and corresponding value
- if root is null, set root \leftarrow new node
- current node \leftarrow root
- for each *bit* in the key
 - if bit == 0,
 - if left child of current node is null, create a new node and attach as the left child
 - move to left child
 - either recursively or by setting current \leftarrow current.left
 - if bit == 1,
 - if right child of current node is null, create a new node and attach as the right child
 - move to right child
 - either recursively or by setting current \leftarrow current.right
- insert corresponding value into current node

RST example

Insert:

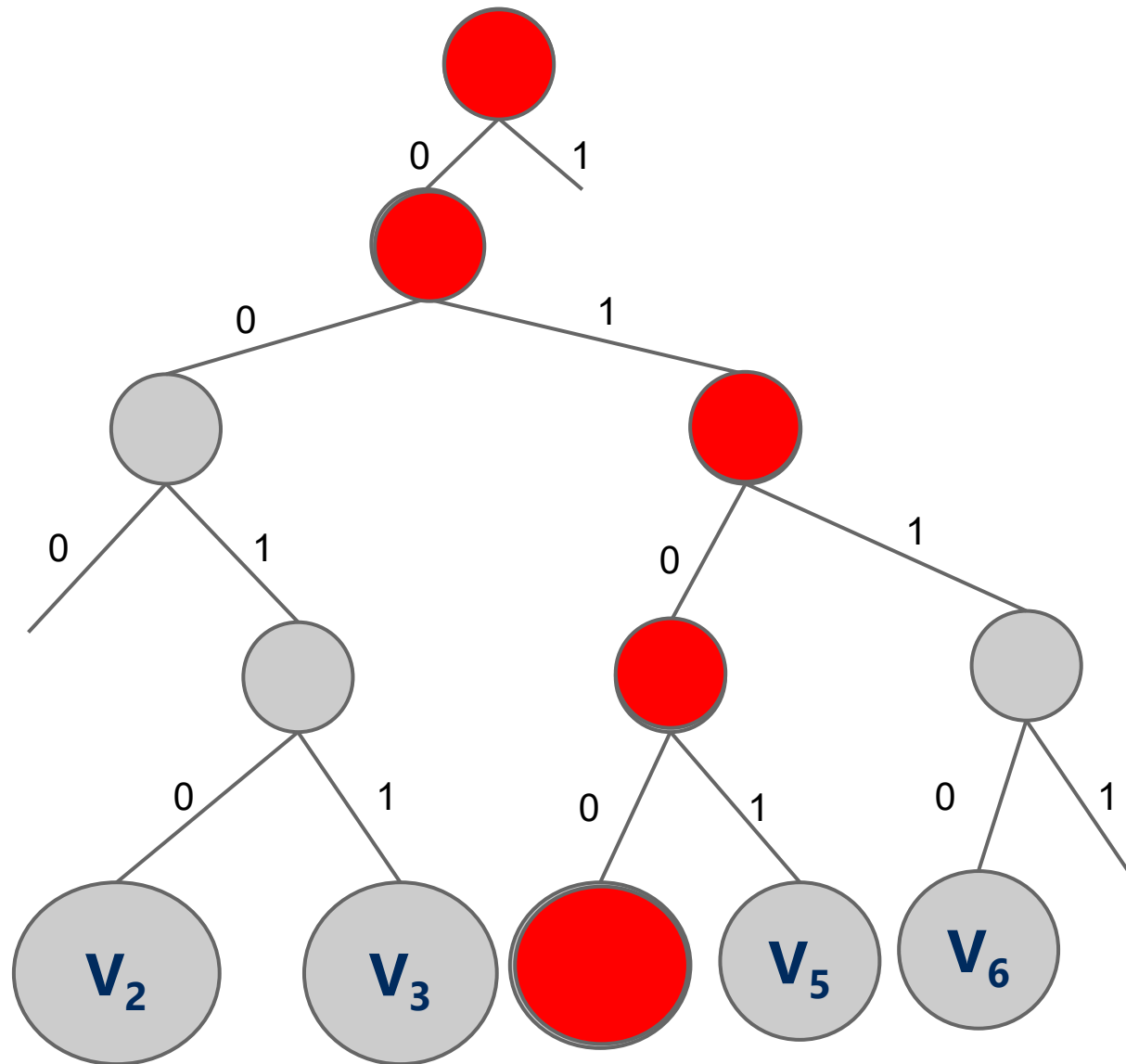
4 0100

3 0011

2 0010

6 0110

5 0101



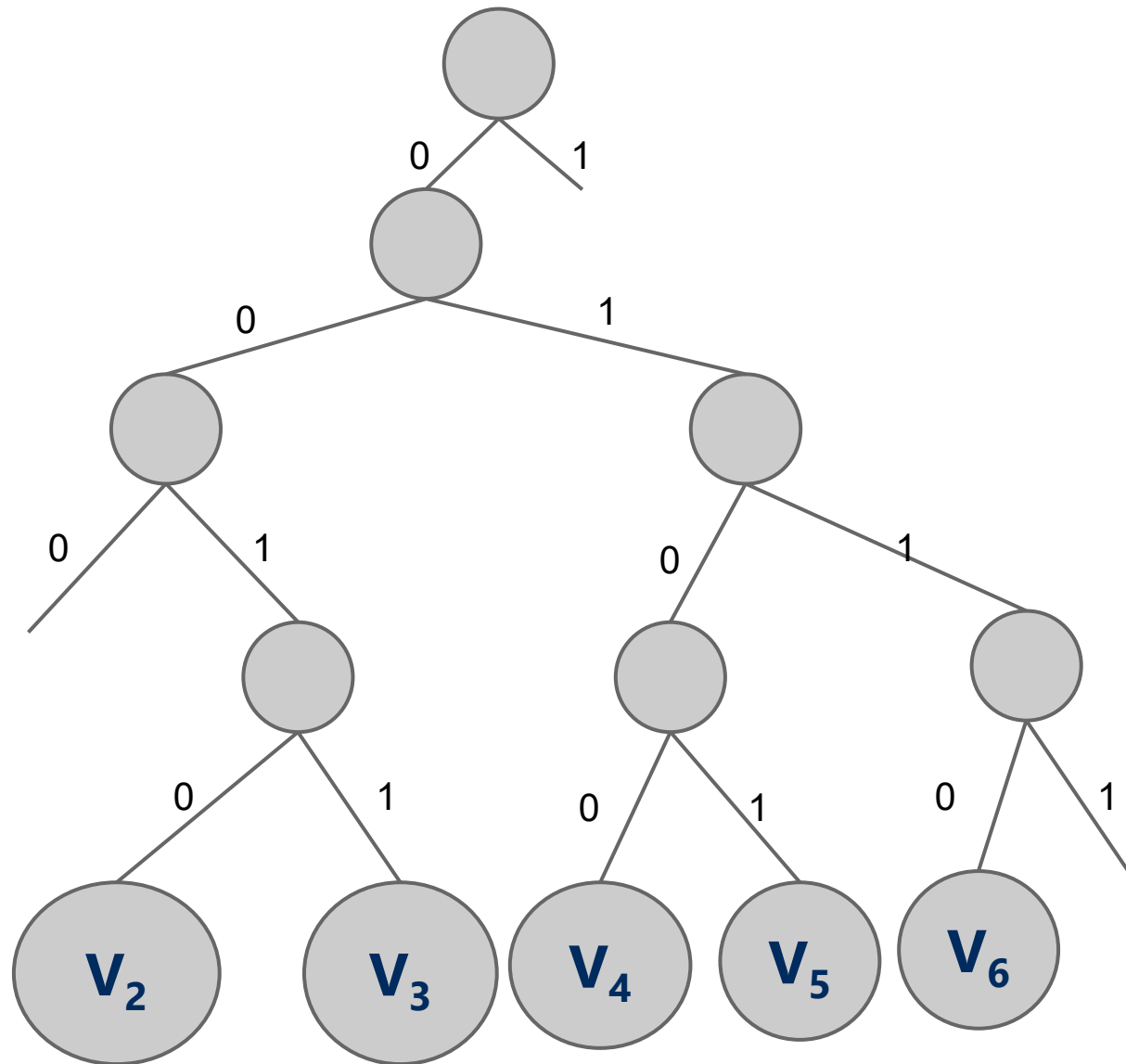
Searching in Radix Search Trie (RST)

- Input: key
- current node \leftarrow root
- for each *bit* in the key
 - if current node is null, return *key not found*
 - if bit == 0,
 - move to left child
 - either recursively or by setting current \leftarrow current.left
 - if bit == 1,
 - move to right child
 - either recursively or by setting current \leftarrow current.right
- if current node is null or the value inside is null
 - return *key not found*
- else return the value stored in current node

RST example

Search:

3 0011

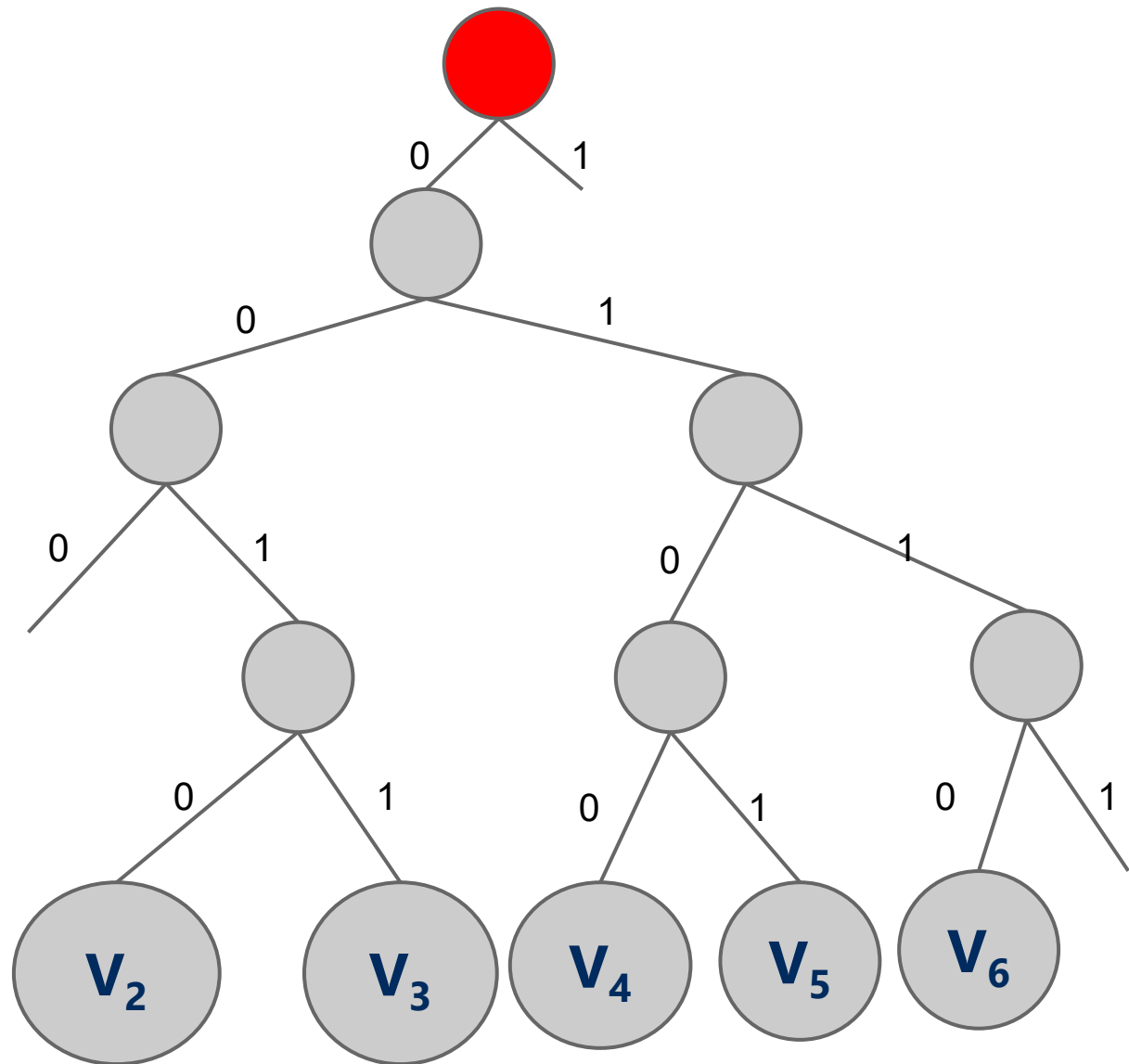


RST example

Search:

3 0011

7 0111

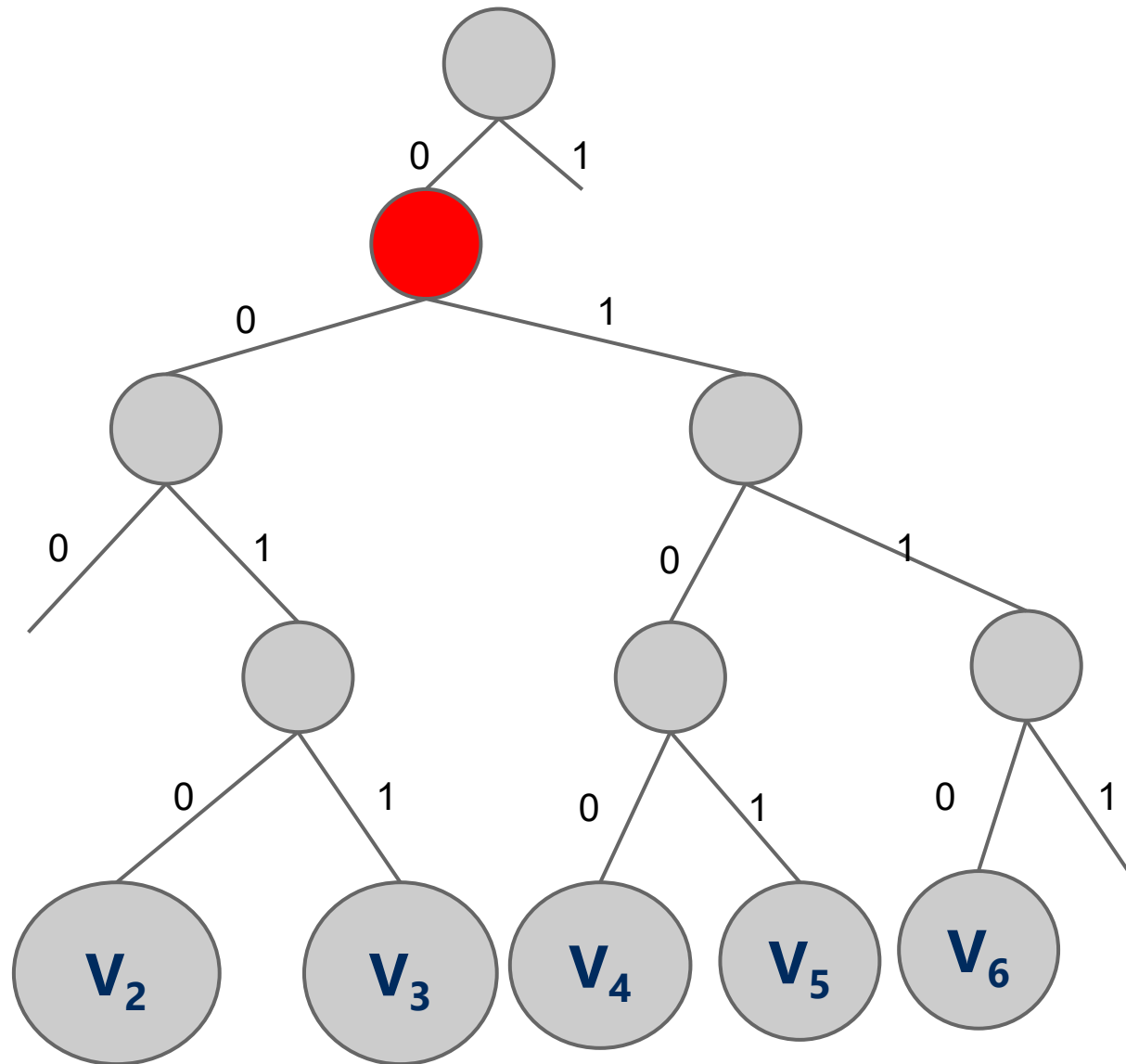


RST example

Search:

3 0011

7 0111

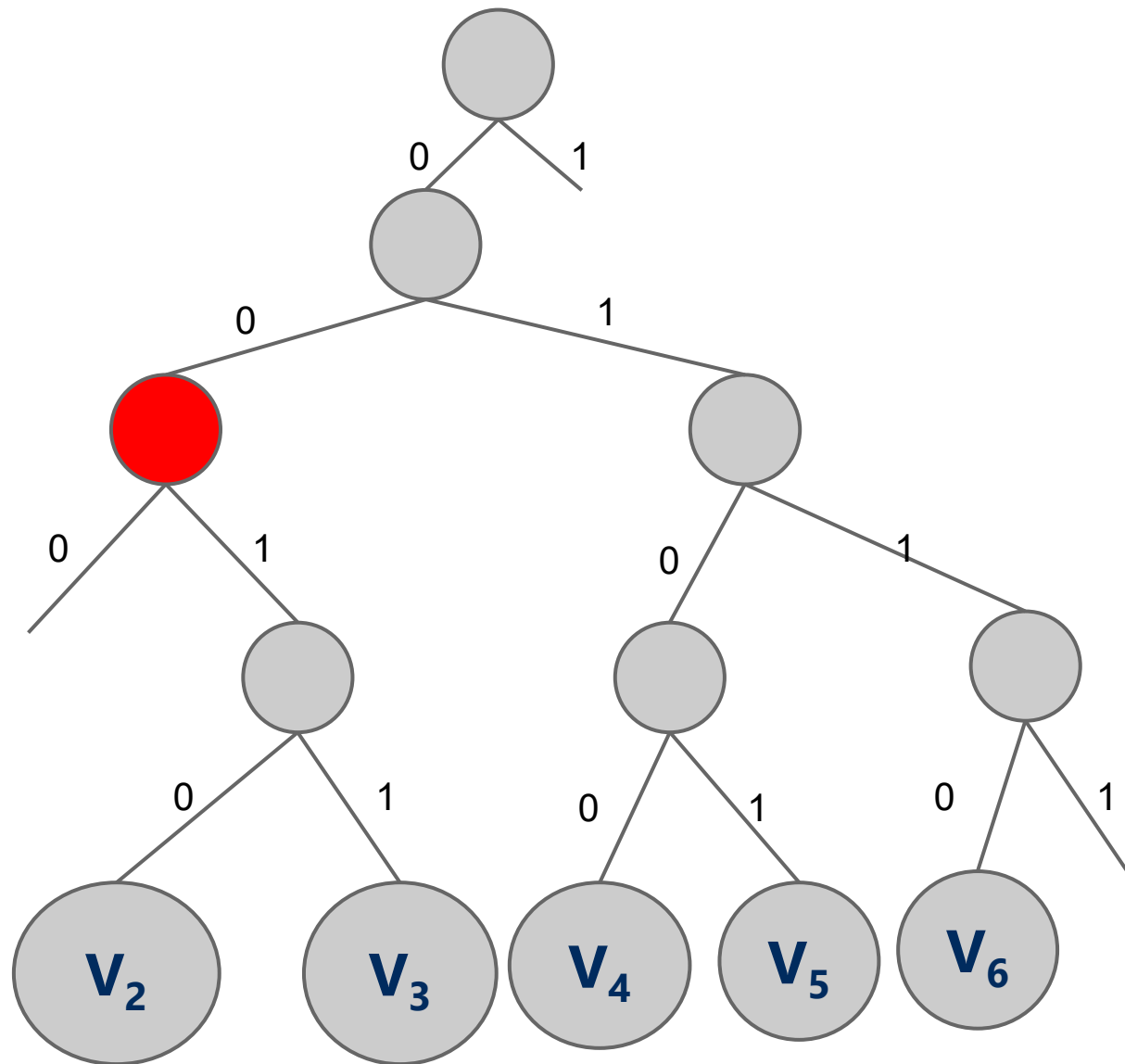


RST example

Search:

3 00**1**1

7 0111

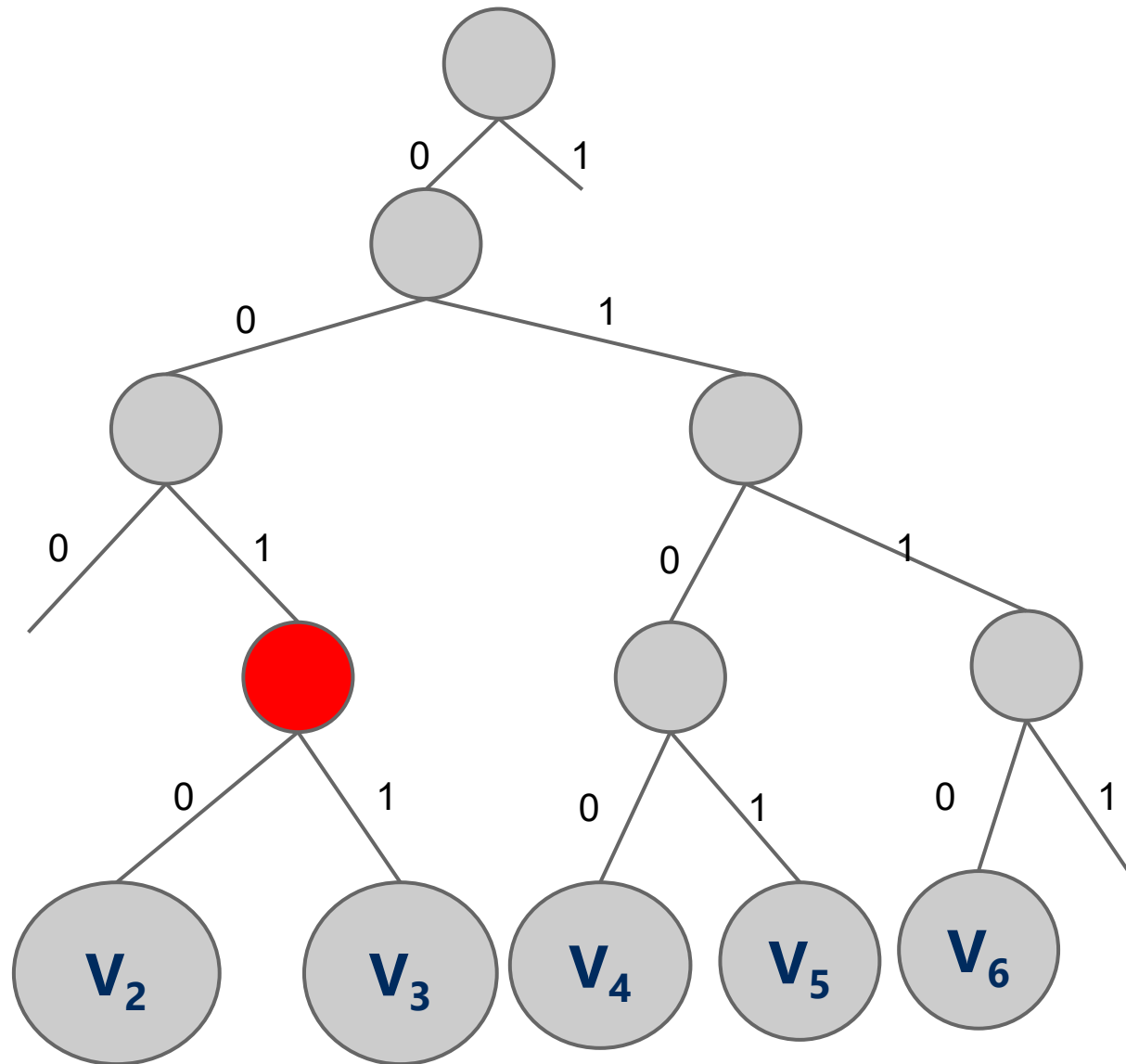


RST example

Search:

3 001**1**

7 0111

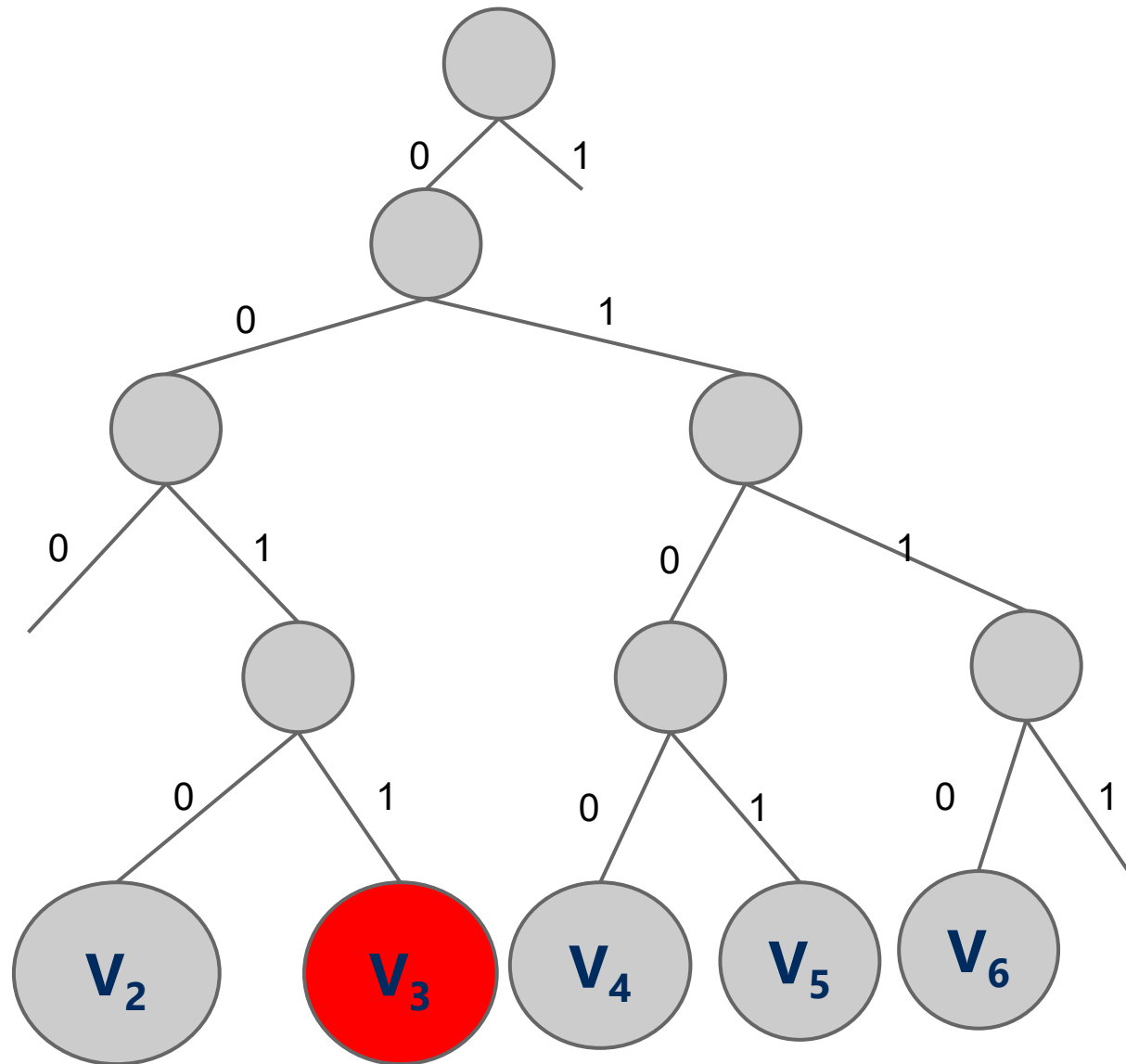


RST example

Search:

3 0011

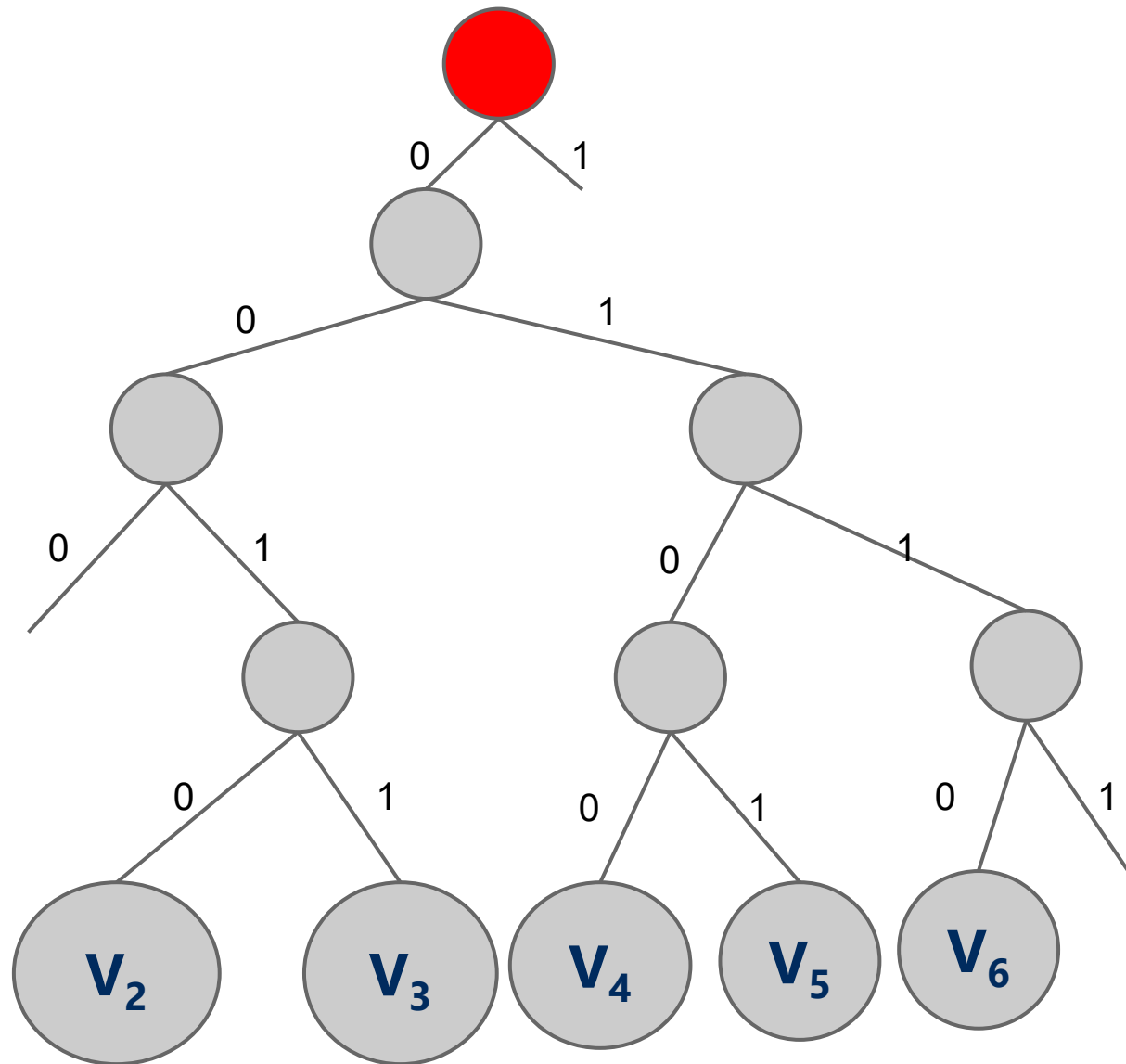
7 0111



RST example

Search:

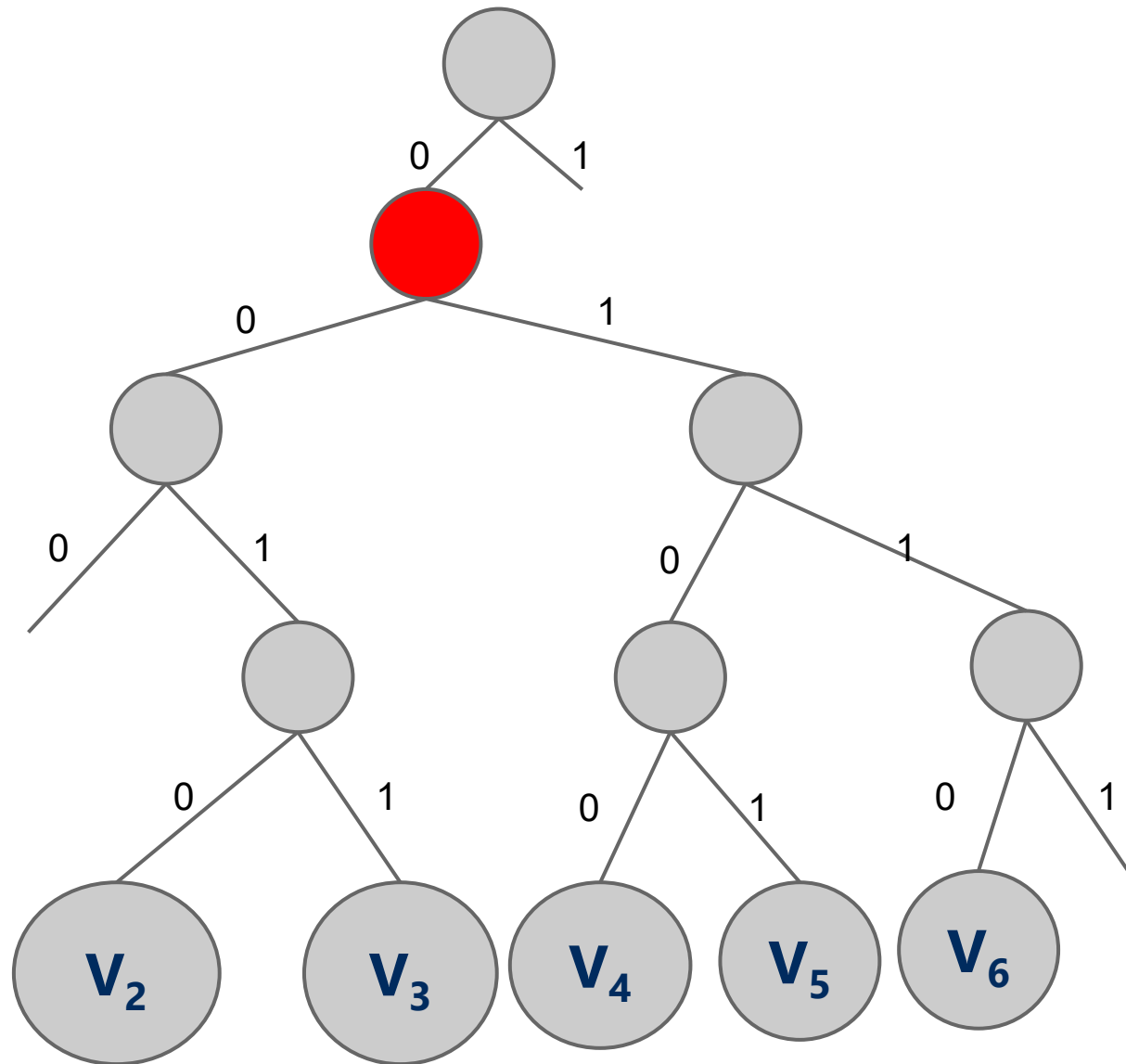
7 0111



RST example

Search:

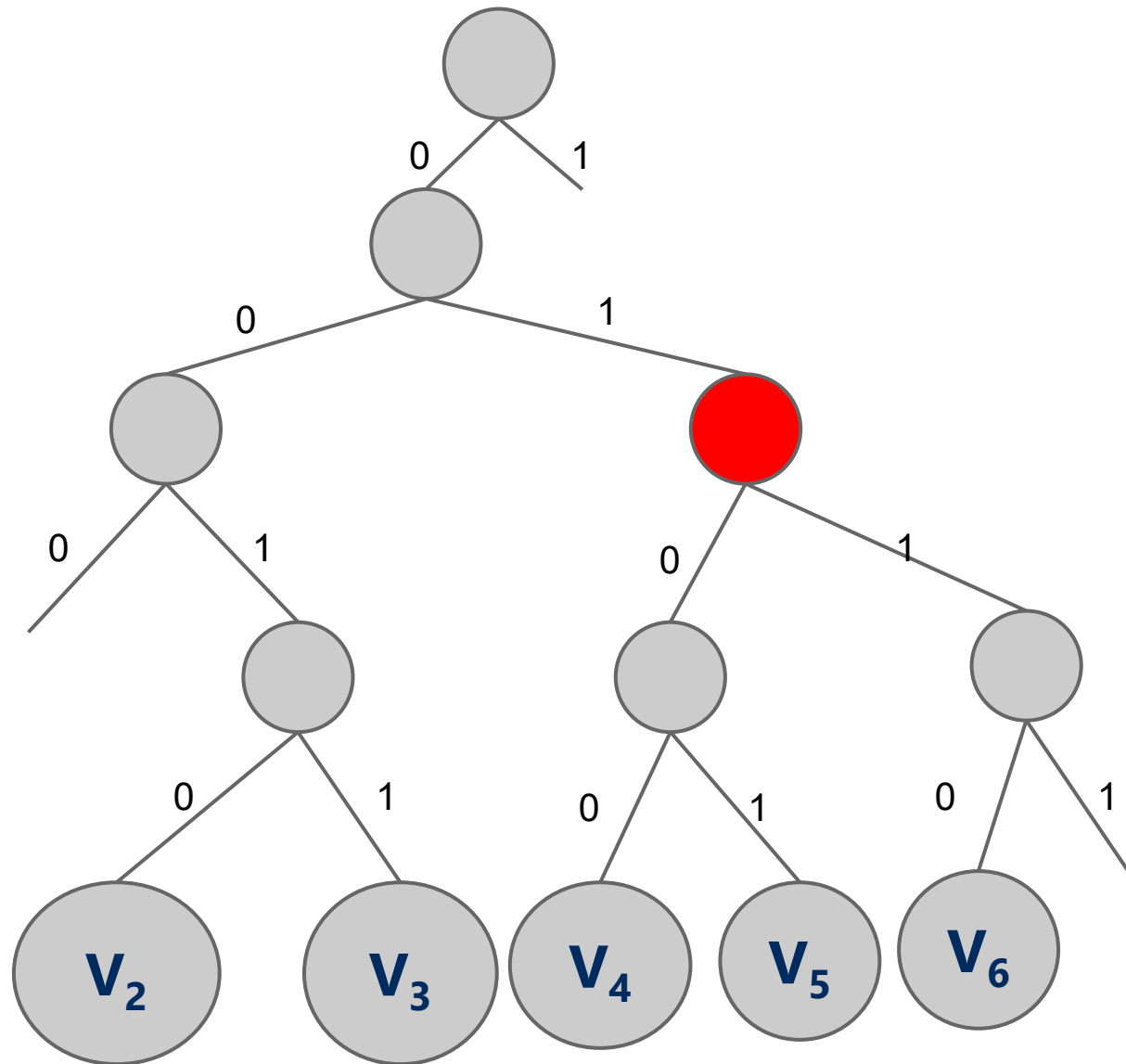
7 0**1**11



RST example

Search:

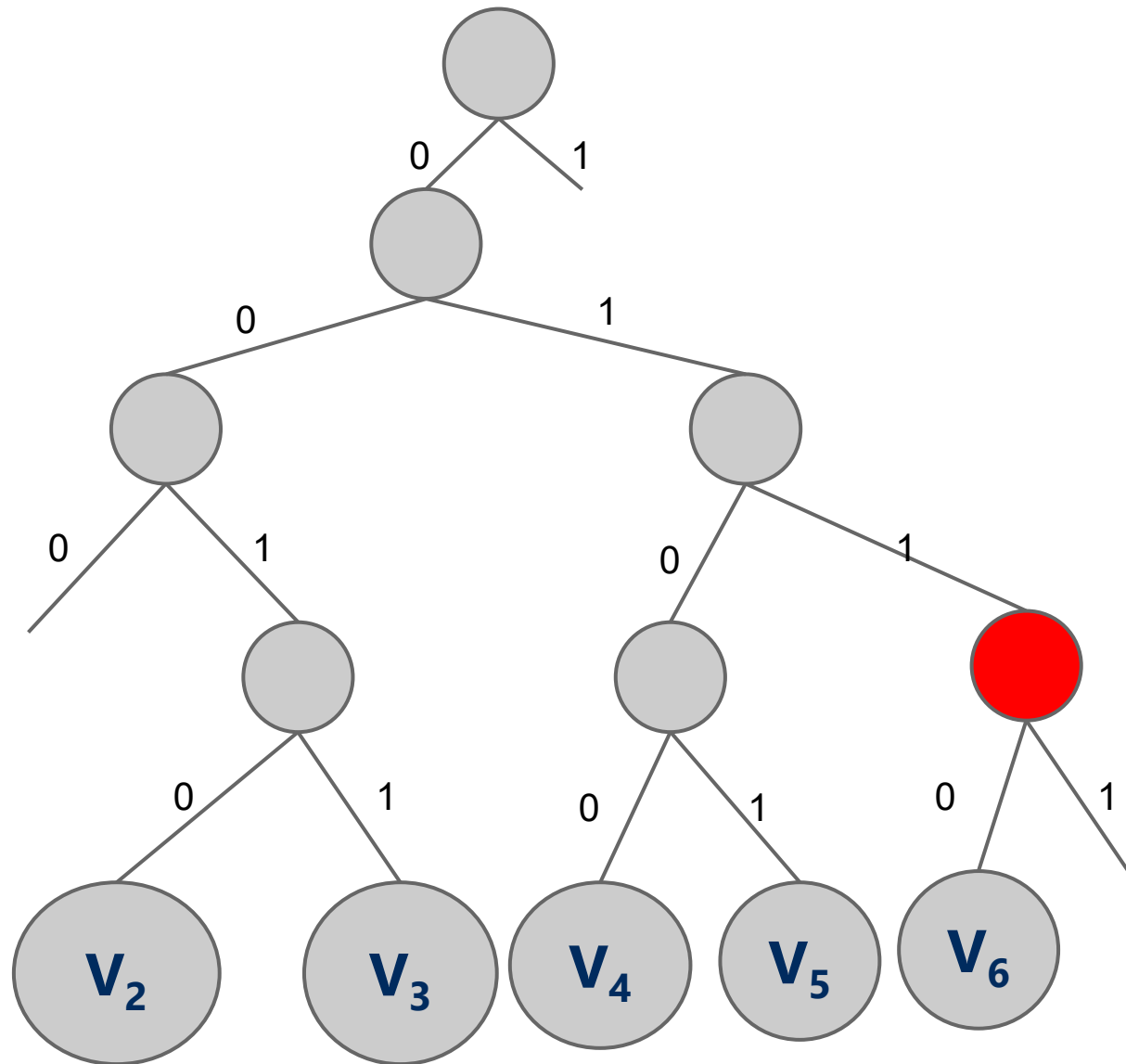
7 0111



RST example

Search:

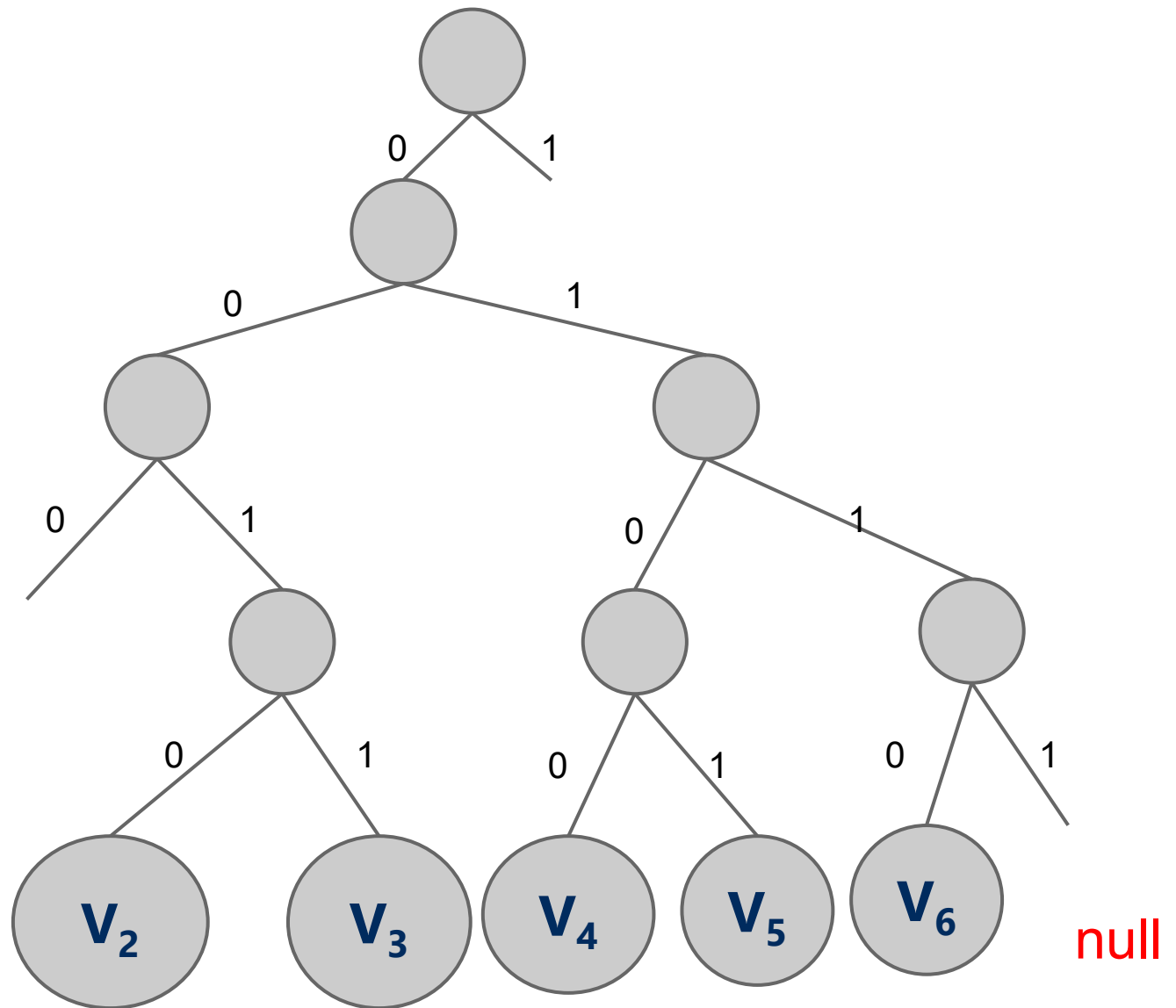
7 0111



RST example

Search:

7 0111



RST analysis

- Runtime?
- $O(b)$, the bit length of the key
 - However, this time we don't have full key comparisons
- Would this structure work as well for other key data types?
- Characters?
 - Characters are the same as 8-bit ints (assuming simple ascii)
- Strings?
- May have huge bit lengths
- How to store Strings?

Larger branching factor tries

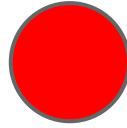
- In our binary-based Radix search trie, we considered one bit at a time
- What if we applied the same method to characters instead of bits in a string?
 - What would this new structure look like?
 - How many children per node?
 - up to R (the alphabet size)
 - Also called R -way radix search tries

Adding to R-way Radix RST

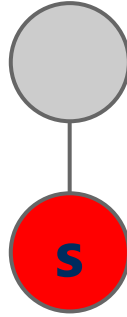
- if root is null, set root \leftarrow new node
- current node \leftarrow root
- for *each character c* in the key
 - *Find the cth child*
 - if child is null, create a new node and attach as the cth child
 - move to child
 - either recursively or by current \leftarrow child
- if at last character of key, insert value into current node

Another trie example

she

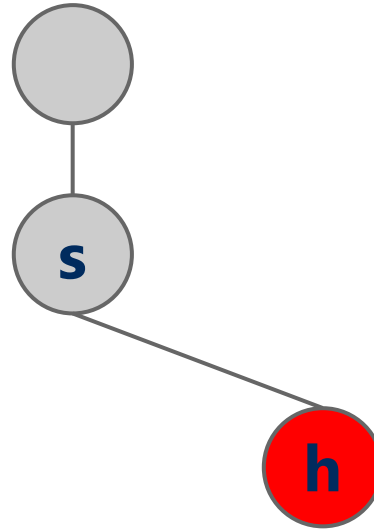


Another trie example



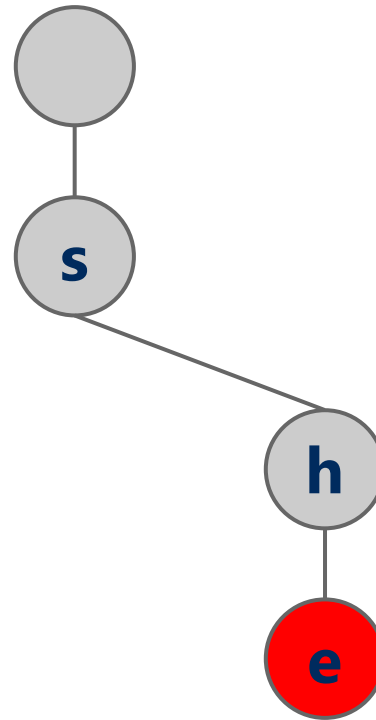
she

Another trie example



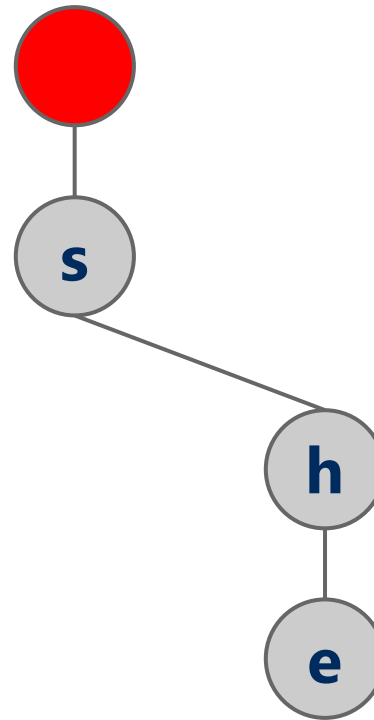
she

Another trie example



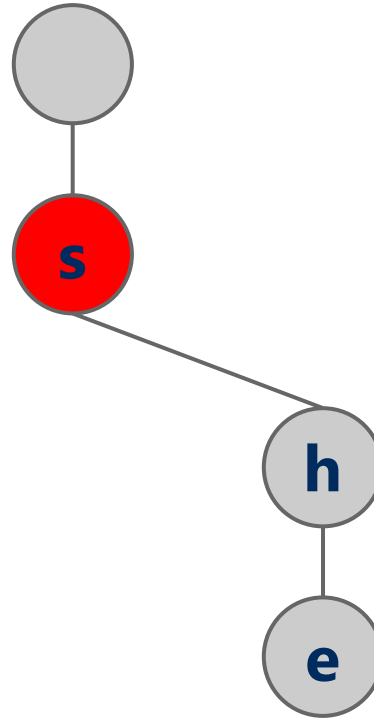
she

Another trie example



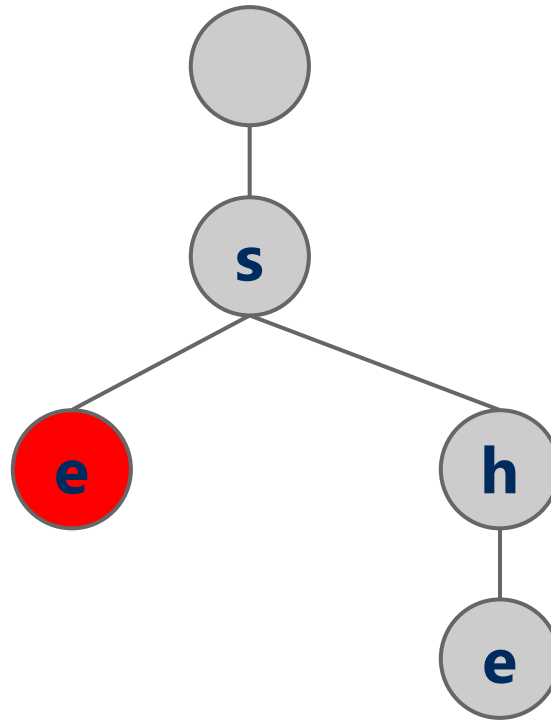
sell

Another trie example



sell

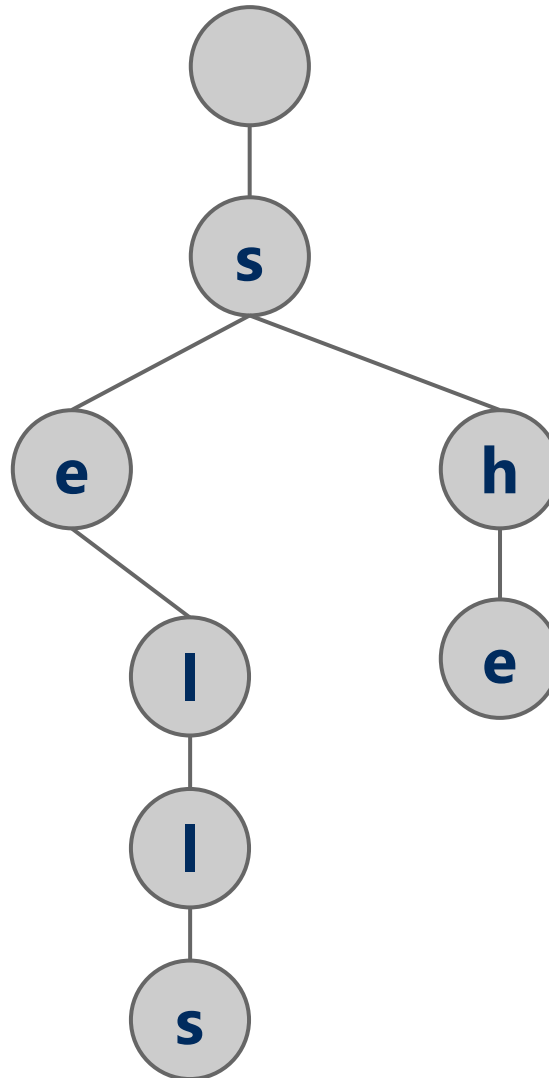
Another trie example



se|ls

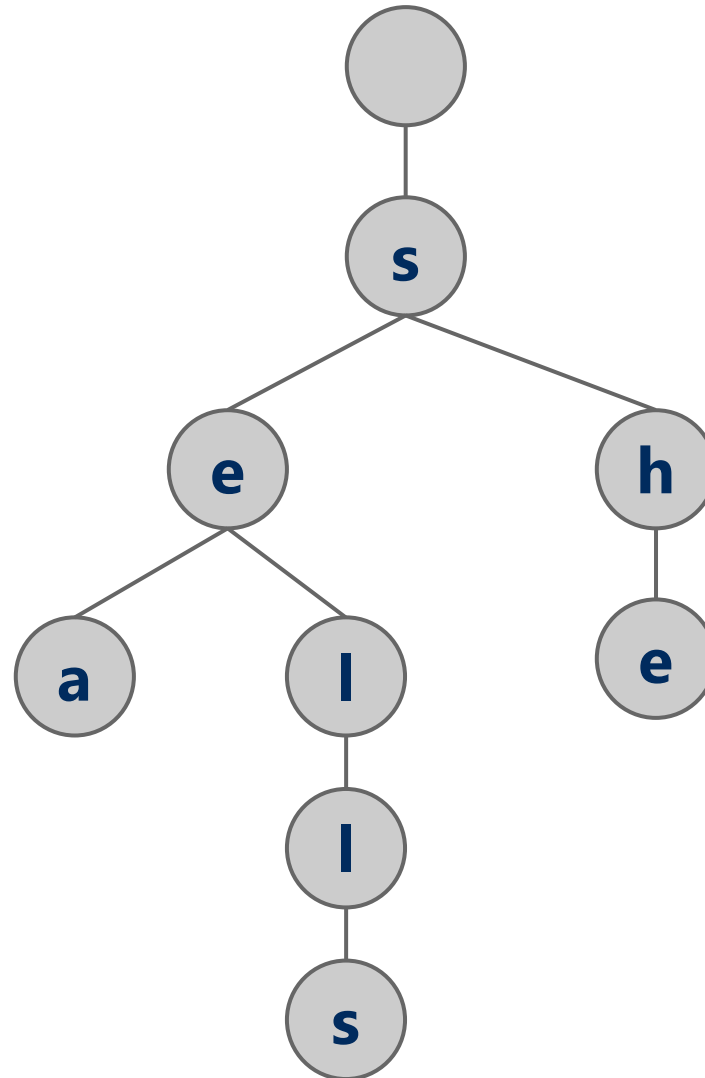
Another trie example

sells



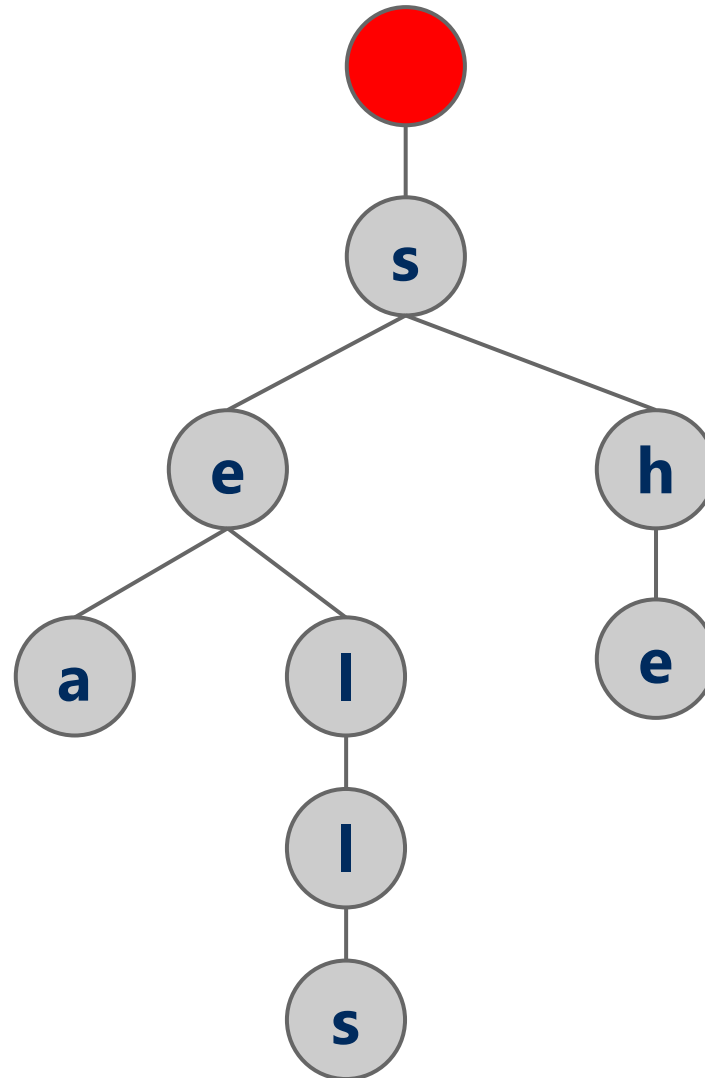
Another trie example

sea



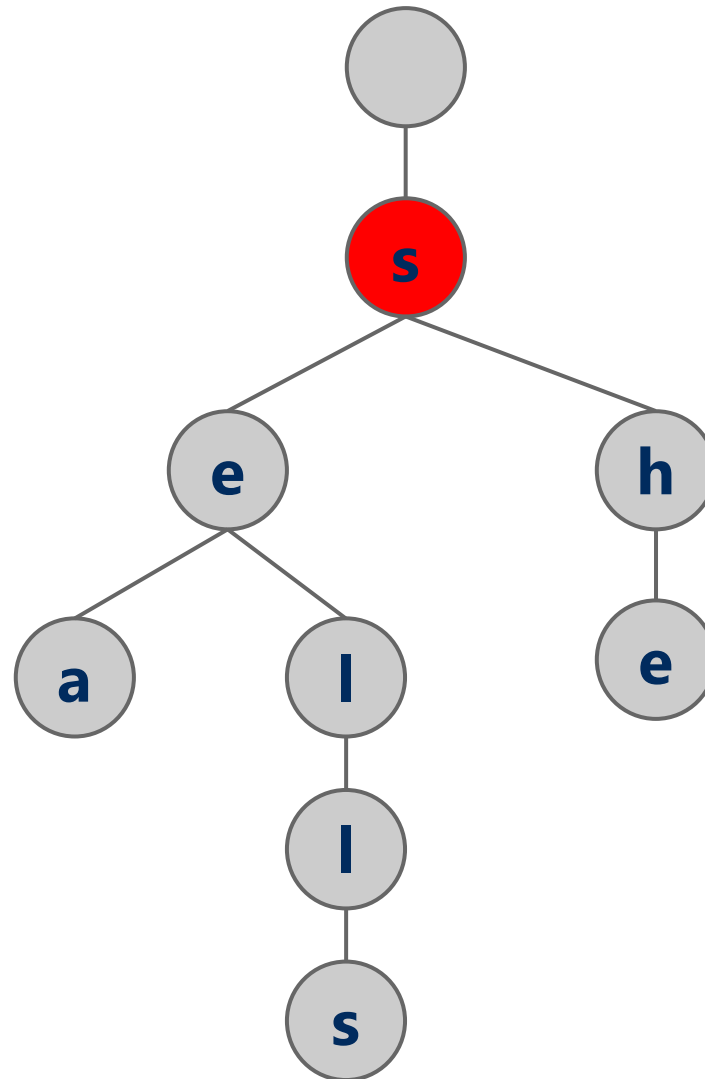
Another trie example

shells



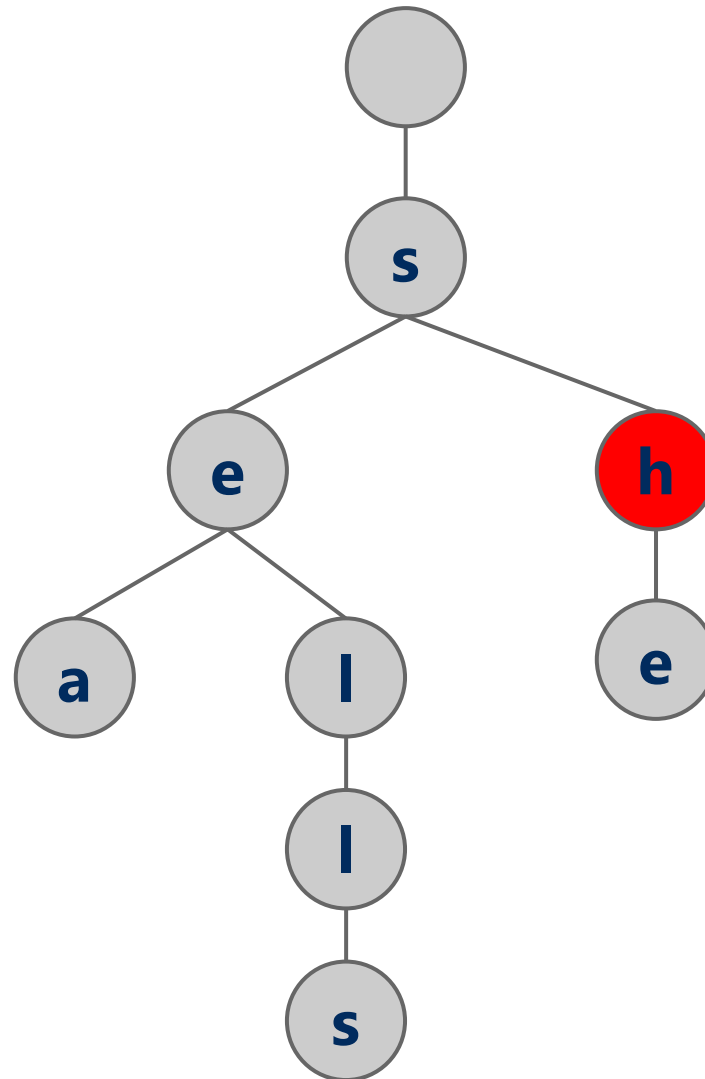
Another trie example

shells



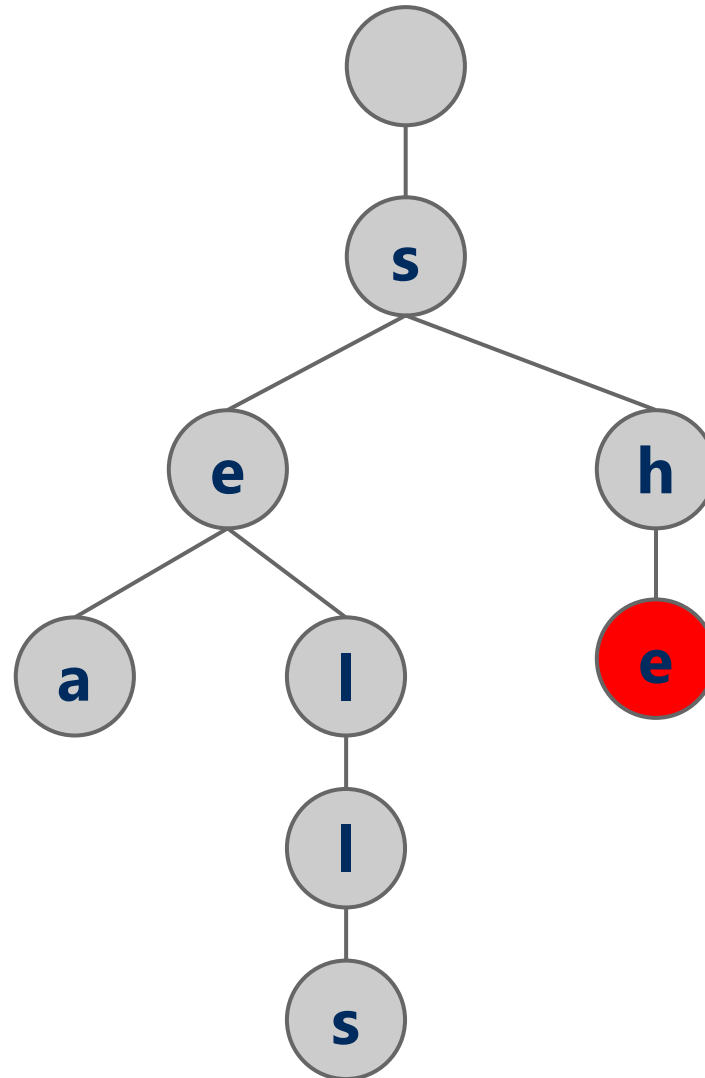
Another trie example

shells



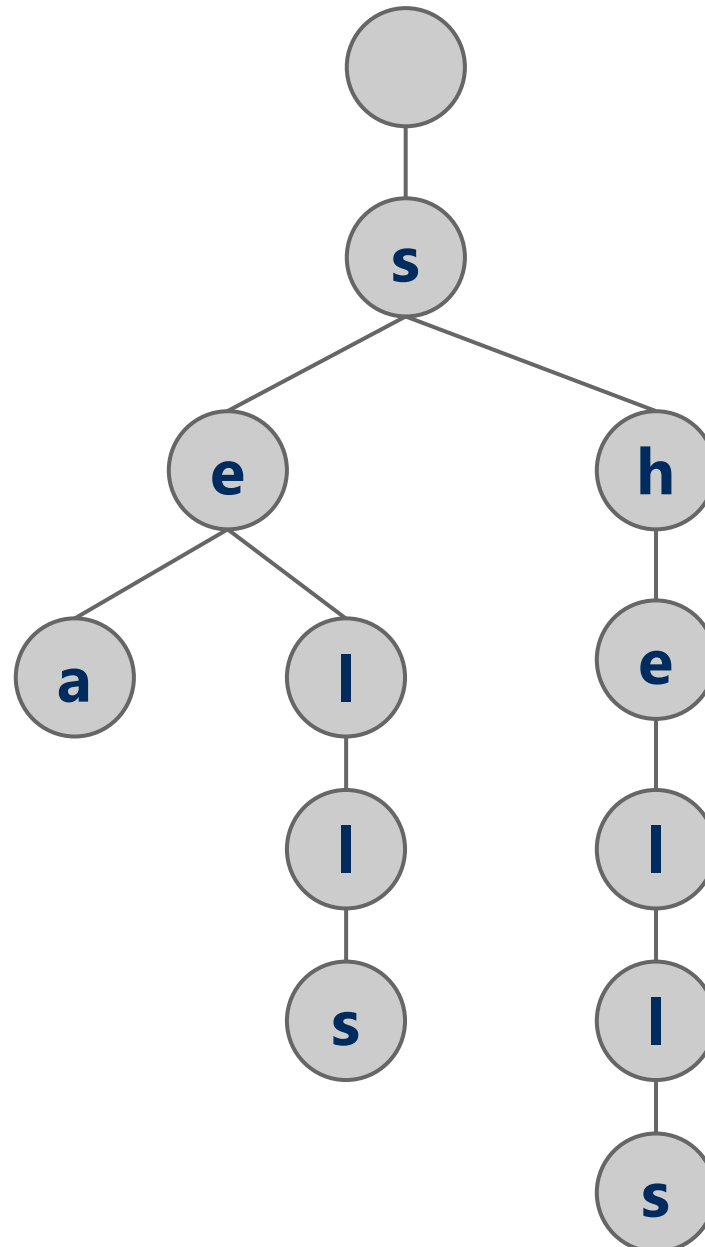
Another trie example

shells

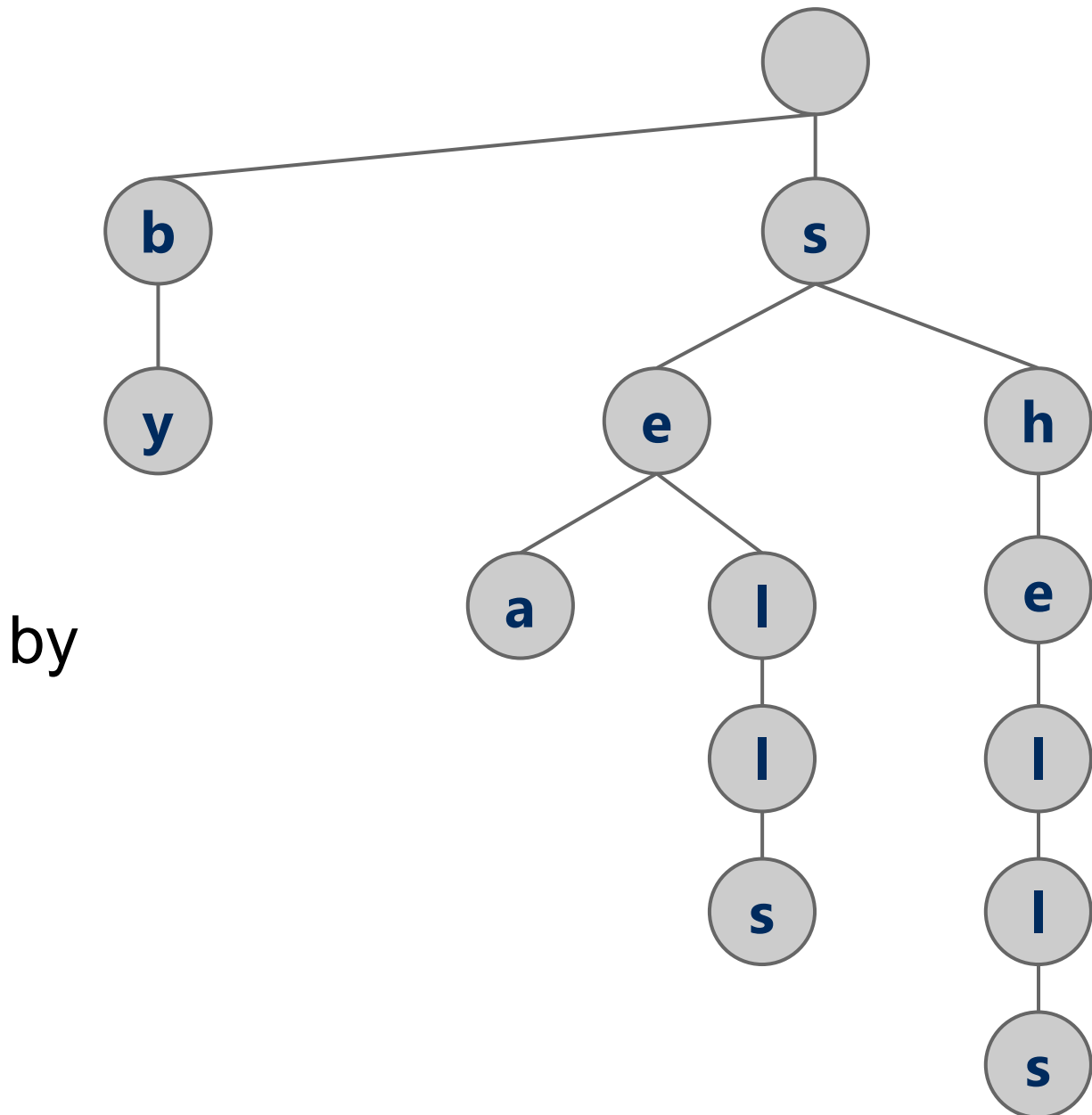


Another trie example

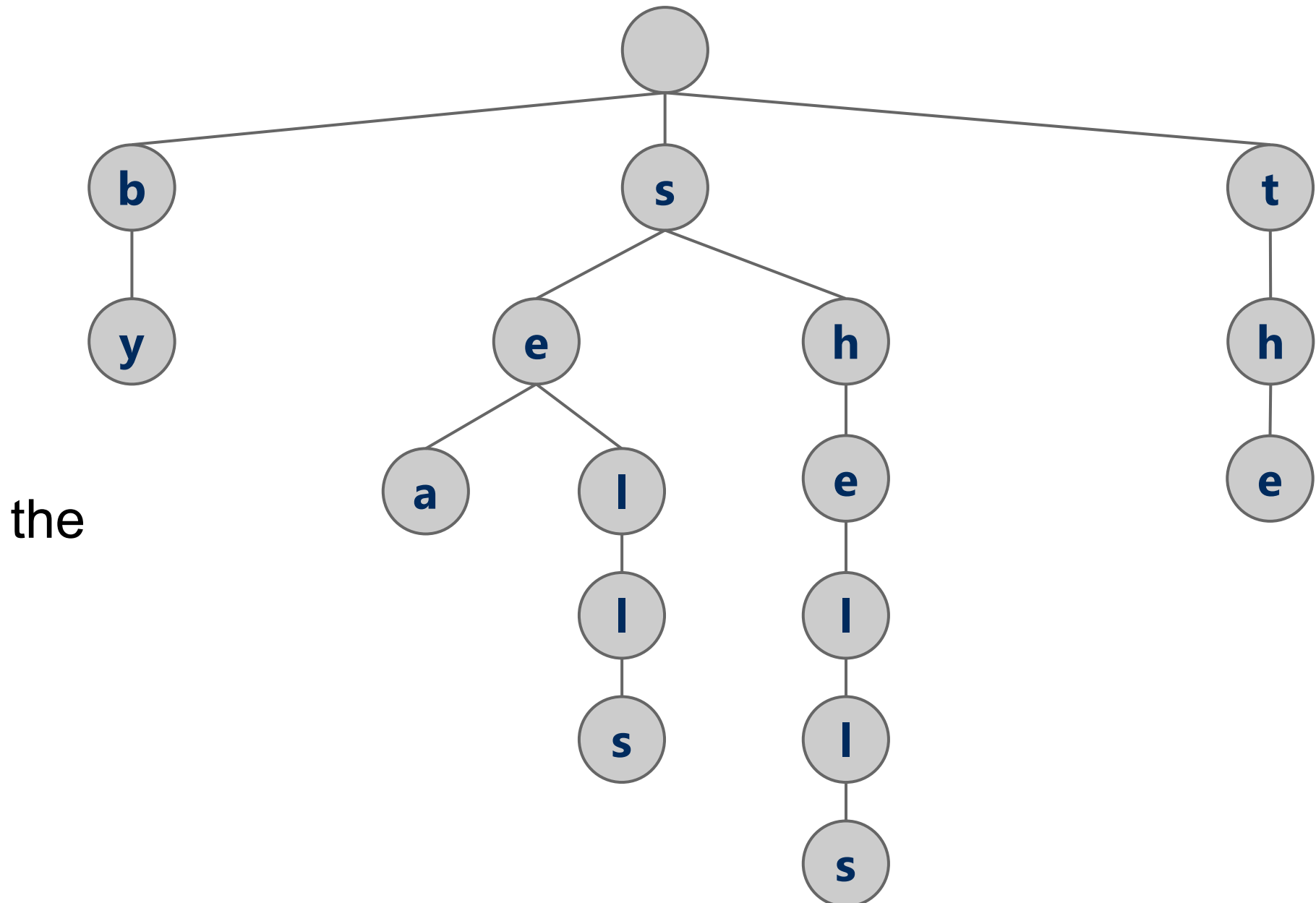
shells



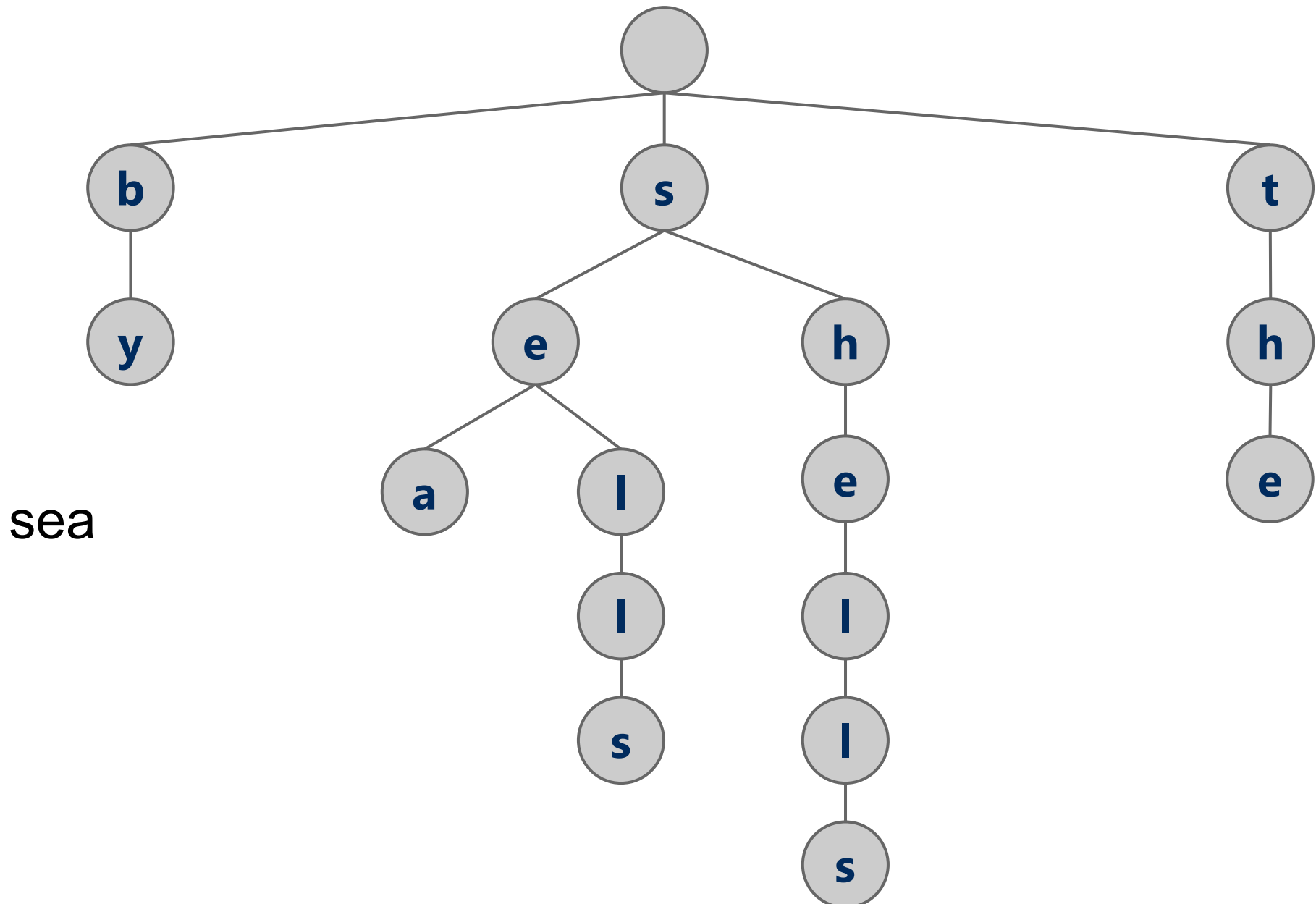
Another trie example



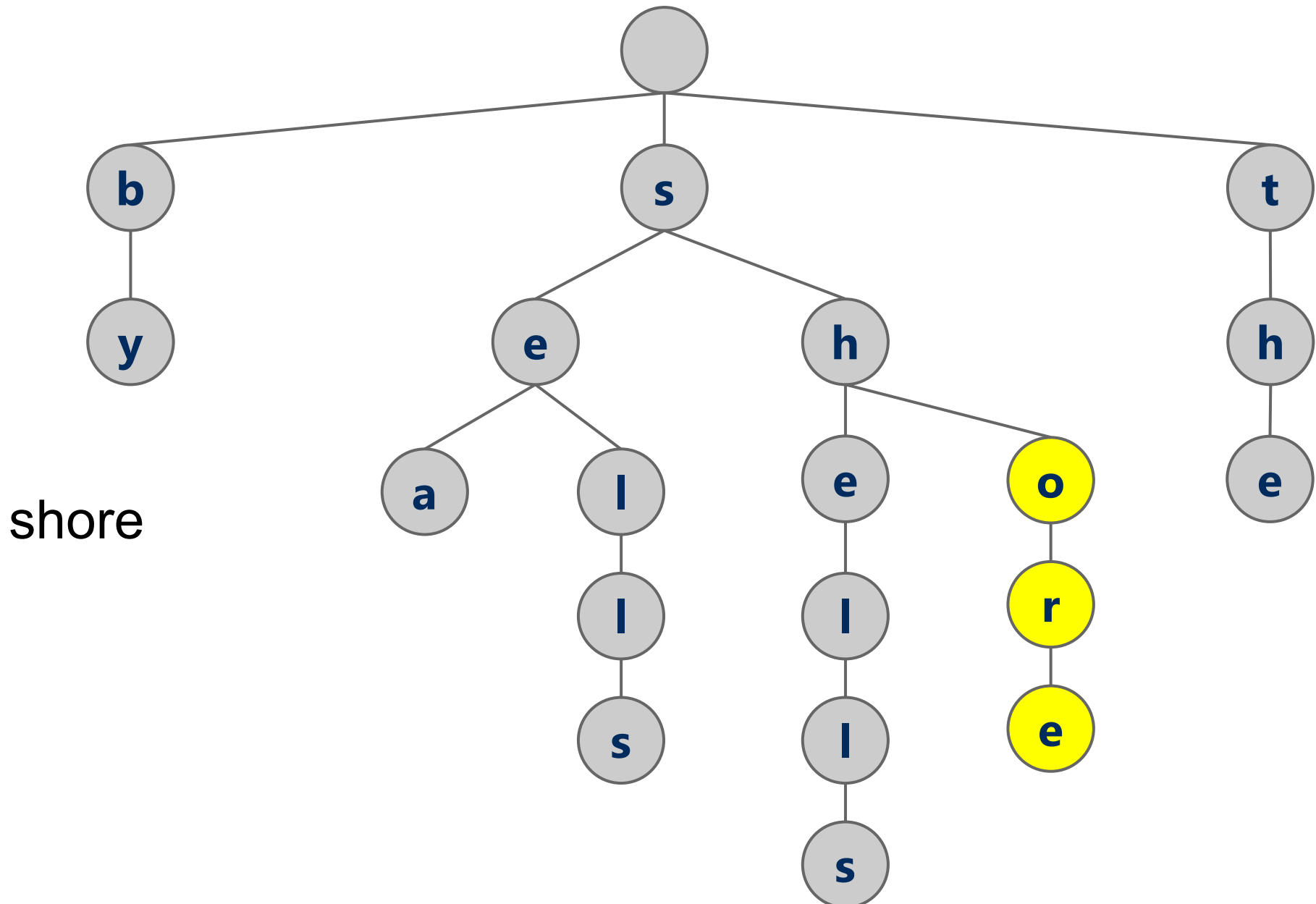
Another trie example



Another trie example



Another trie example



Analysis

- Runtime of add and *search hit*?
- $O(w)$ where w is the character length of the string
 - So, what do we gain over RSTs?

- $w < b$

- e.g., assuming fixed-size encoding

$$w = \frac{b}{\lceil \log R \rceil}$$

- tree height is reduced

Search Miss

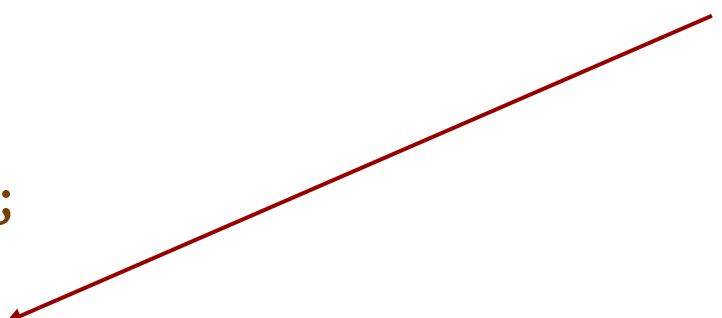
- Search Miss time for R-way RST
 - Require an average of $\log_R(n)$ nodes to be examined
 - Proof in Proposition H of Section 5.2 of the text
- Average tree height with 2^{20} keys in an RST?
 - $\log_2 n = \log_2 2^{20} = 20$
- With 2^{20} keys in a large branching factor trie, assuming 8-bits at a time?
 - $\log_R n = \log_{256} 2^{20} = \log_{256} (2^8)^{2.5} = \log_{256} 256^{2.5} = 2.5$

Implementation Concerns

- See TrieSt.java
 - Implements an R-way trie
- Basic node object:

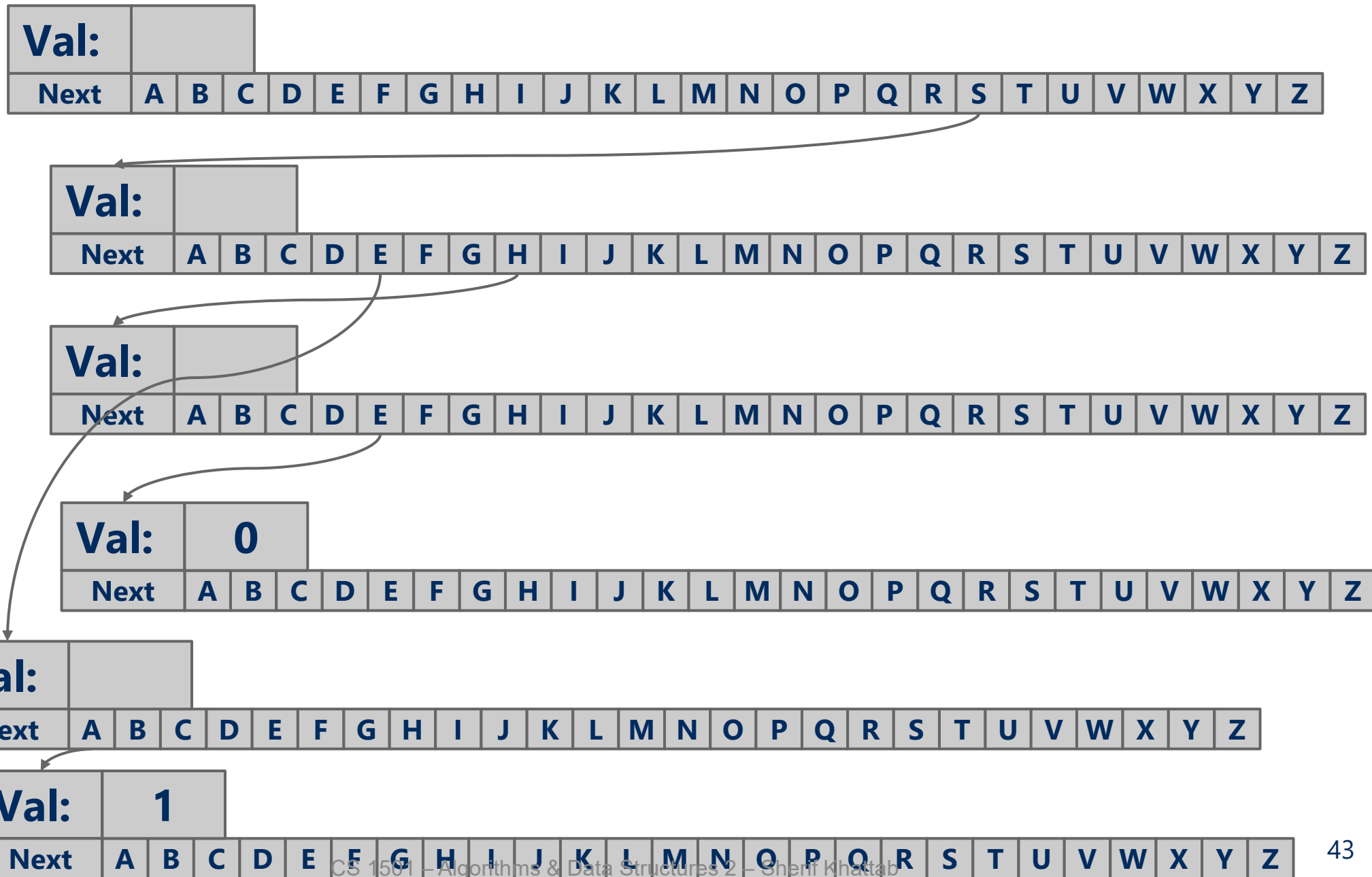
Where R is the branching factor

```
private class Node {  
    private Object val;  
    private Node[] next;  
    private Node(){  
        next = new Node[R];  
    }  
}
```



- Non-null **val** means we have traversed to a valid key
- Again, note that keys are not directly stored in the trie at all

R-way trie example

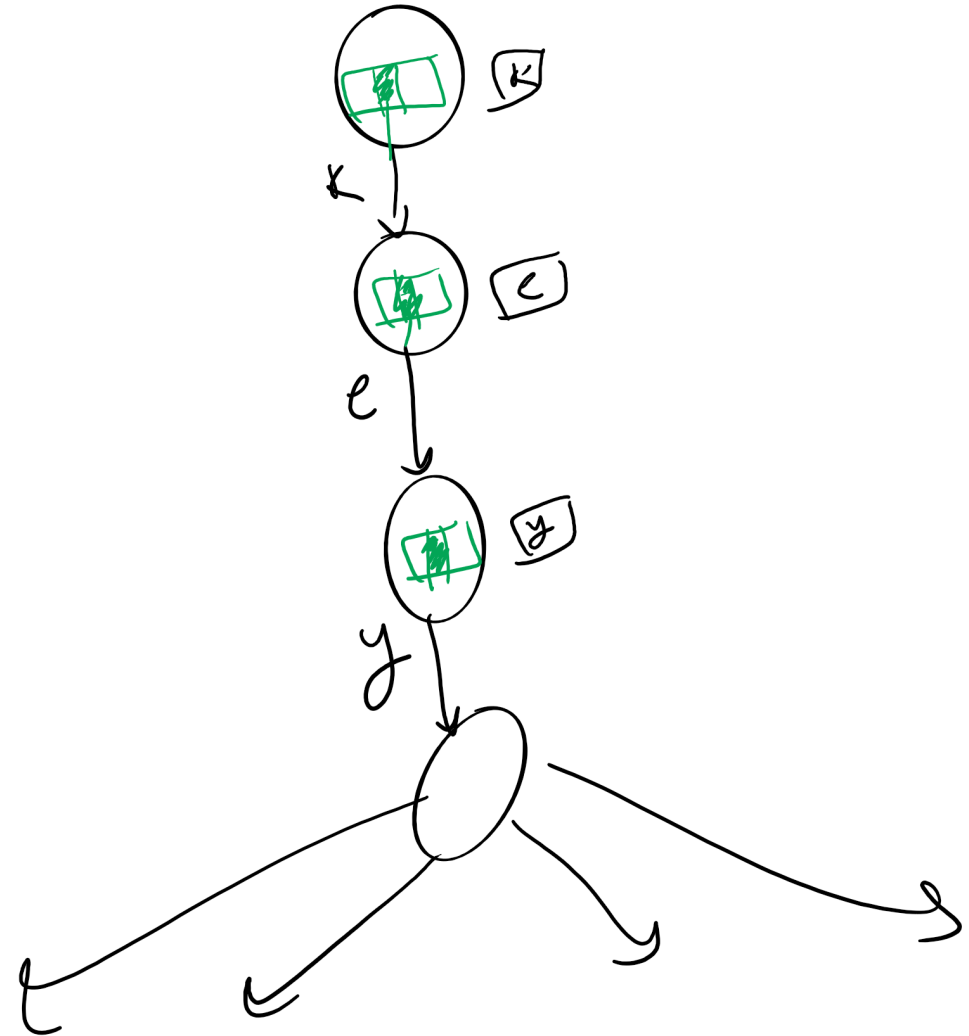


Summary of running time

	insert	Search hit	Search miss
binary RST	$\Theta(b)$	$\Theta(b)$	$\Theta(\log_2 n)$ on average
multi-way RST	$\Theta(w)$	$\Theta(w)$	$\Theta(\log_R n)$

R-way RST's nodes are large!

- Considering 8-bit ASCII, each node contains 2^8 references!
- This is especially problematic as in many cases, a lot of this space is wasted
 - Common paths or prefixes for example, e.g., if all keys begin with "key", that's 255×3 wasted references!
 - At the lower levels of the trie, most keys have probably been separated out and reference lists will be sparse



Solution: De La Briandais tries (DLBs)

Main idea: replace the array inside the node of the R-way trie with a linked-list

DLB Nodelets

Two alternative implementations:

```
private class DLBNode {  
    private Object val;  
    private T character;  
    private Node sibling;  
    private Node child;  
}
```

If search terminates on a node with non-null value, key is found; otherwise, not found.

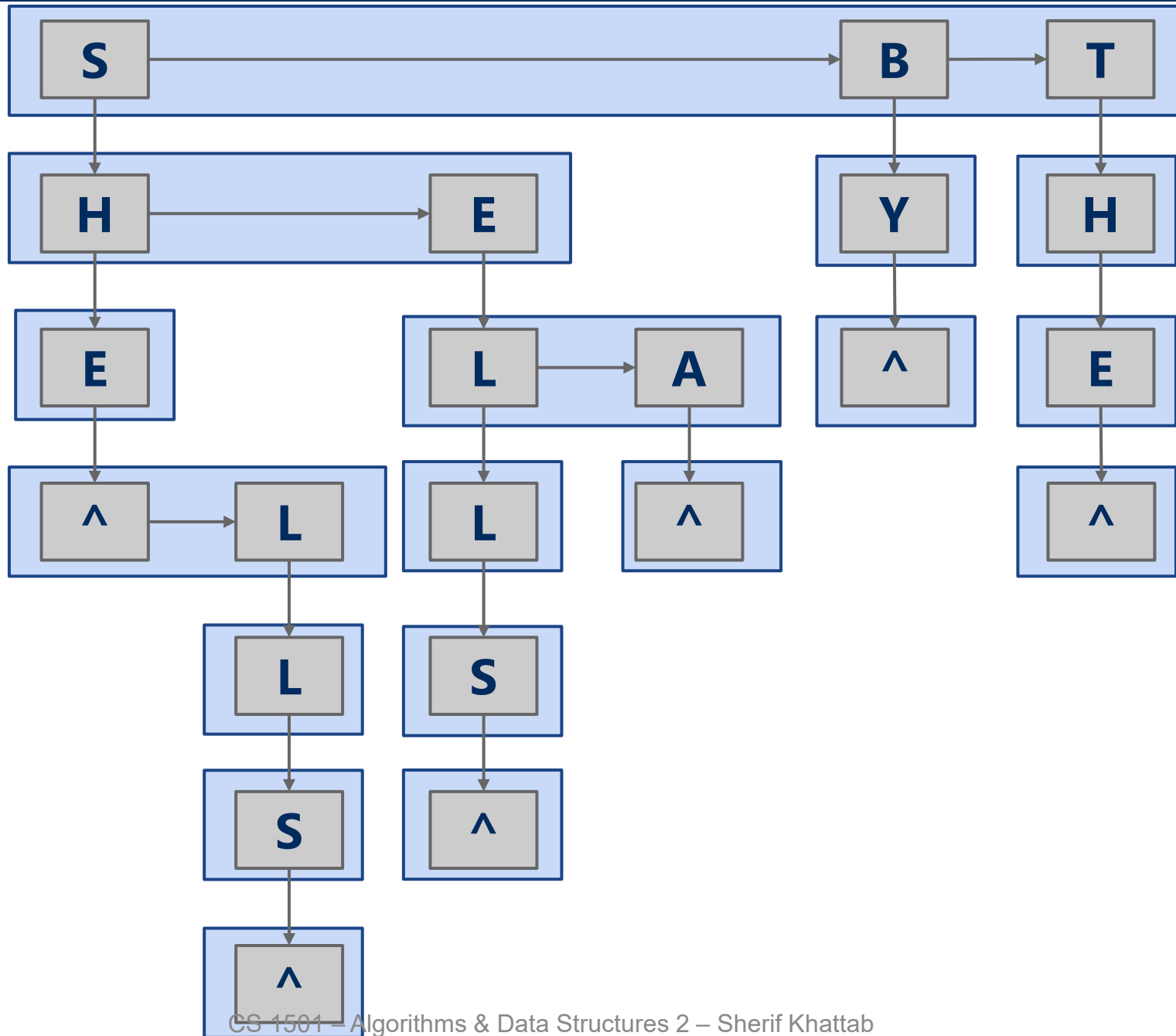
```
private class DLBNode {  
    private Object val;  
    private Character character;  
    private Node sibling;  
    private Node child;  
}
```

Add a sentinel character (e.g., ^) to each key before add and search
If search encounters null, key not found; otherwise, key is found

Adding to DLB Trie

- if root is null, set root \leftarrow new node
- current node \leftarrow root
- for each *character* c in the key
 - Search for c in the linked list headed at current using sibling links
 - if not found, create a new node and attach as a sibling to the linked list
 - move to child of the found node
 - either recursively or by current \leftarrow child
- if at last character of key, insert value into current node and return

DLB Example



DLB analysis

- How does DLB performance differ from R-way tries?
- Which should you use?

	Search hit insert	
R-way RST	$\theta(w)$	
DLB	$\theta(wR)$	

Runtime Comparison for Search Trees/Tries

	Search hit	Search miss <i>(average)</i>	insert
BST	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$
RB-BST	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
DST	$\Theta(b)$	$\Theta(\log n)$	$\Theta(b)$
RST	$\Theta(b)$	$\Theta(\log n)$	$\Theta(b)$
R -way RST	$\Theta(w)$	$\Theta(\log n)$	$\Theta(w)$
DLB	$\Theta(w \cdot R)$	$\Theta(\log_{\frac{R}{w}} n)$	$\Theta(w \cdot R)$

Final notes on Search Tree/Tries

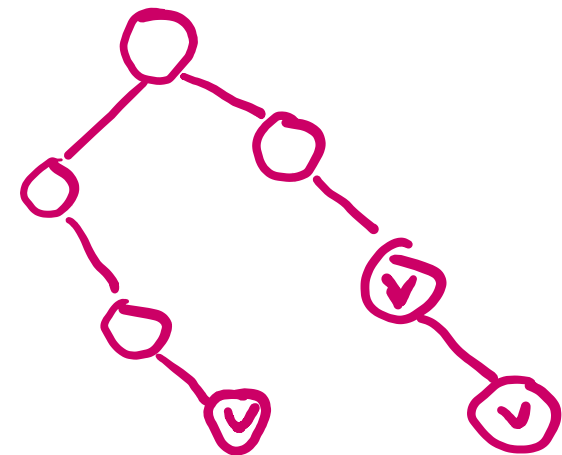
- We did not present an exhaustive look at search trees/tries, just the sampling that we're going to focus on
- Many variations on these techniques exist and perform quite well in different circumstances
 - Ternary search Tries
 - R-way tries without 1-way branching
- See the table at the end of Section 5.2 of the text

Muddiest Points

- **Q: When creating a DST, does it have to start handling keys starting with the leftmost bit, or can it also handle them by starting with the rightmost bit?**
- The algorithm can go either way on the bitstring of the key as long as the direction is the same for all operations

Muddiest Points

- **Q: Would a trie be able to contain a value with less bits than the root, and if so how?**
- In a trie, none of the nodes (including the root) contains any key
- If the question is “can a trie contain keys of different bit lengths?”,
 - the answer is yes
 - Interior nodes have non-null values in that case
 - The trie here has three keys
 - 011
 - 111
 - 11



Muddiest Points

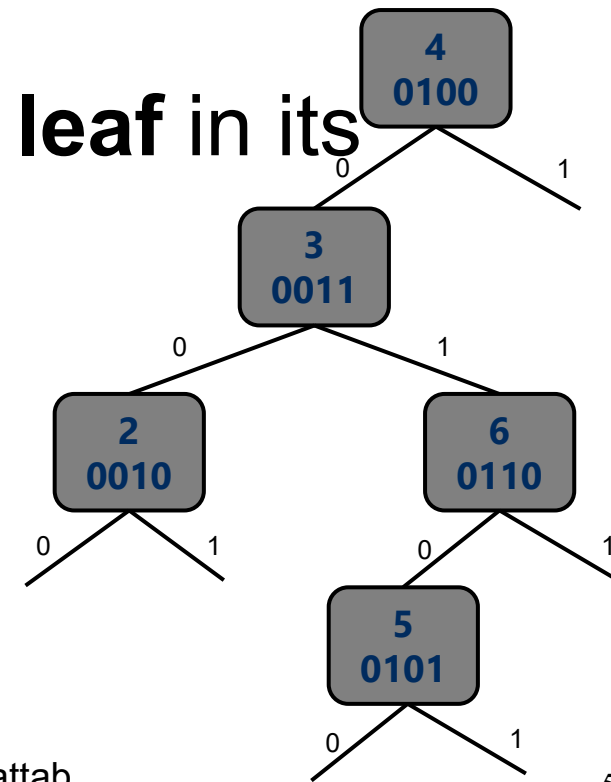
- **Q: How is a Trie different from a Red-Black BST?**
- a trie is different from a search tree because trie doesn't store the keys inside its nodes but a search tree does
- More in the next question

Muddiest Points

- **Q: when would you use a DST or an RST?**
- **Q: What's the application of RST?**
- DST and RST are efficient in checking if a target key is a prefix of any of the keys in the tree
 - e.g., making routing decisions in the Internet
- DSTs are preferred over BSTs when bits of keys are randomly distributed (i.e., the probability of each bit being zero is 0.5)
 - The DST will be balanced in this case without having to use the more complicated Red-Black BST
- RSTs are preferred over BSTs when bit lengths of keys are close to $\log n$
 - The RST will be balanced in this case without having to use the more complicated Red-Black BST
- Note that DST and RST don't provide the extra operations (e.g., predecessor and successor) provided by BST

Muddiest Points

- **Q: how can any node in 3's subtree replace 3 in DST example**
- Because all nodes in 3's subtree share a common prefix with length 1 with 3
 - The node that replaces 3 will still be found using the DST search algorithm
- For simplicity, we replace 3 with any **leaf** in its subtree



Muddiest Points

- **Q: Could you spend more time going through new lecture**
- Sometimes, addressing the muddiest points takes up a large portion of class time
- Usually, new material is embedded between the muddiest points responses

Muddiest Points

- **Q: Is the insertion position for DST based on the first bit that is different from the last insert? Or is it based on the relative comparison to last insert?**
- DST Add Algorithm for adding a key k and a corresponding value
 - if root is null, add k at the root and return
 - $\text{current} \leftarrow \text{root}$
 - if k is equal to the current node's key, replace value and return
 - if current bit of k is 0,
 - if left child is null, add k as left child
 - else continue to left child
 - if current bit of k is 1,
 - if left child is null, add k as right child
 - else continue to right child

Muddiest Points

- **Q: When is DST preferable to radix search trie?**
- **A: When bit lengths of keys are $\gg \log n$**

Muddiest Points

- **Q: I don't understand the advantage of making another node in the DLB instead of the tree structure.**
- **A: DLB saves space when the number of children per node in an R-way RST is small**

Muddiest Points

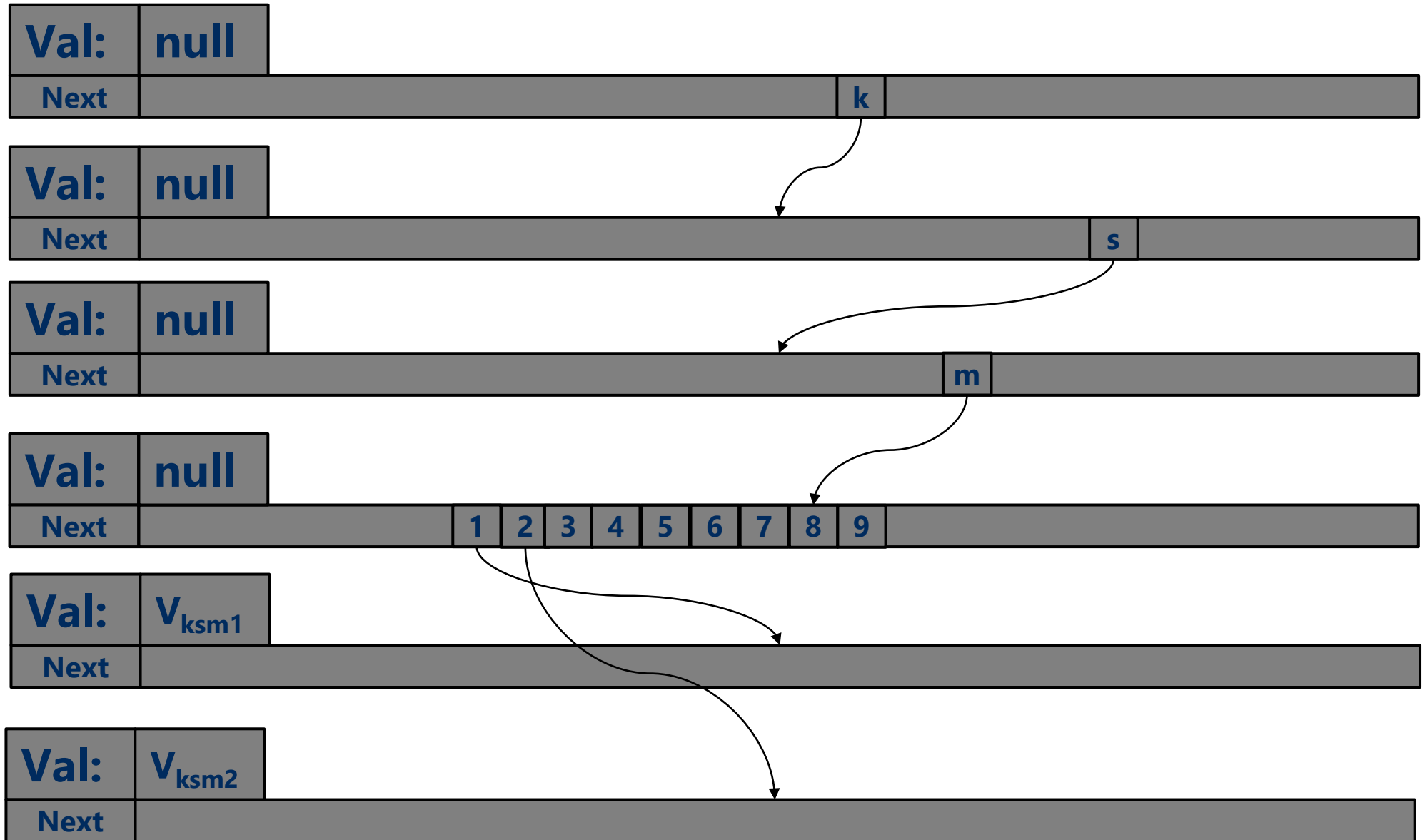
- **Q: How does DLB save space over r way trie? Example please?**
- Let's the set of keys:
 - ksm1 ... ksm9
- How big does an 256-way RST take vs. a DLB trie?

R-way RST

```
private class Node {  
    private Object val;  
    private Node[] next;  
  
    private Node(){  
        next = new Node[R];  
    }  
}
```

Each node takes $4 \cdot (R+1) = 4 \cdot 257 = 1028$ bytes,
assuming 4 bytes per reference variable

R-way RST



R-way RST

We will end up with $4 + 9 = 13$ nodes

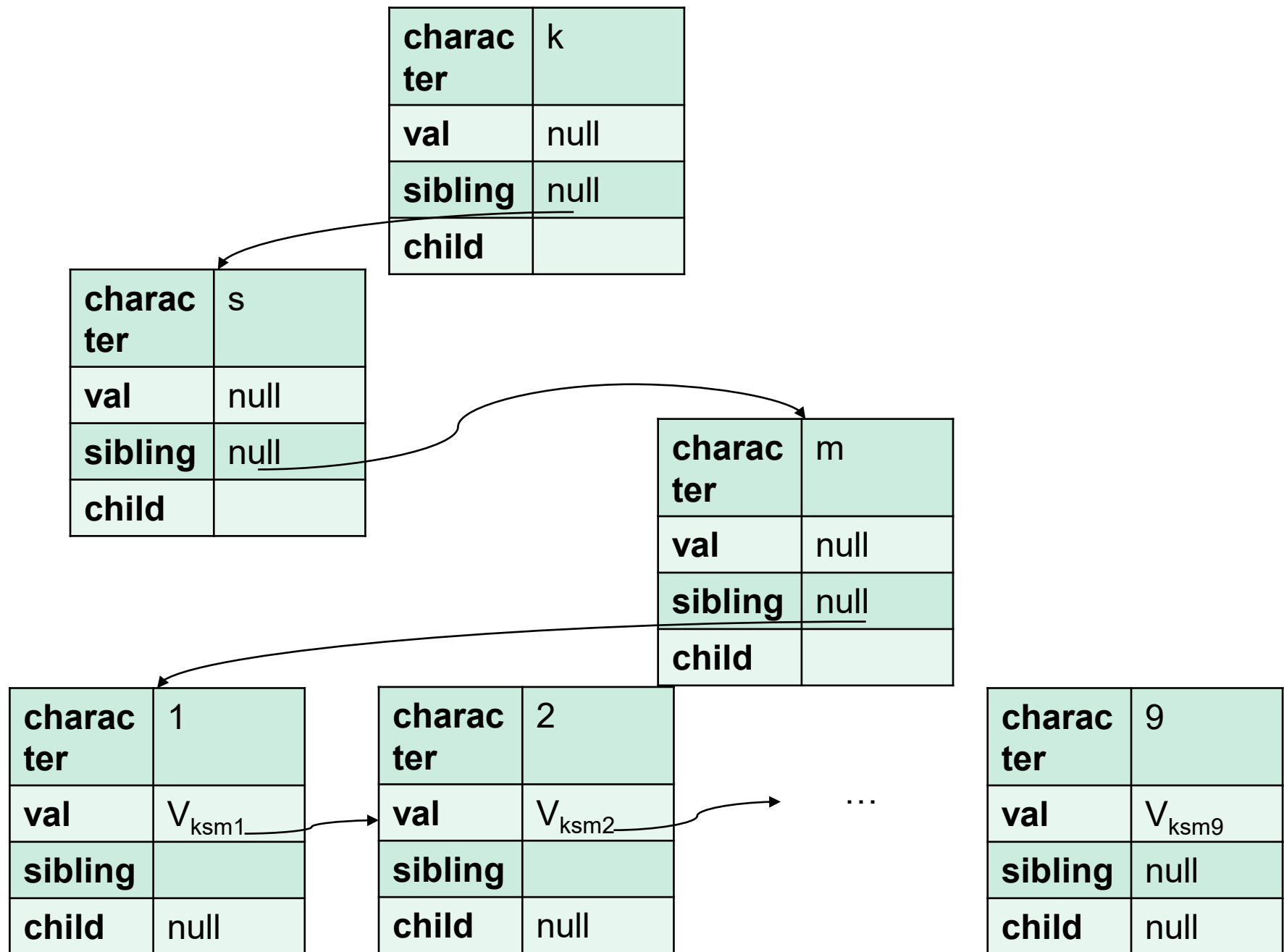
Total space is $13 * 1028 = 13,364$ bytes

DLB Trie

```
private class DLBNode<T> {  
    private Character character;  
    private Object val;  
    private Node sibling;  
    private Node child;  
}
```

Each node takes $4 \times 4 = 16$ bytes, assuming 4 bytes per reference variable

DLB Trie



DLB Trie

We will end up with 12 nodes

Total space is $12 * 16 = 192$ bytes

Compare to 13,364 bytes with an R-way RST

Muddiest Points

- **Q: What determines the number of bits you use for the bit representation of a key in DSTs and RSTs?**
- Typically, the number of bits is determined by the application
 - e.g., keys are Pitt usernames, PeopleSoft IDs, English sentences, etc.
- It is better to re-encode the keys to have a bit length of $\log n$ bits each
 - Requires extra space to store the mapping from old keys to new keys
 - sometimes not possible: e.g., when n is not known in advance
- Better yet, we can assign bit lengths based on frequency of access:
 - Shorter bitstrings for more frequently accessed keys
 - Results in smaller **average** search time

$$\text{average Case runtime} = \sum_{\text{all cases}} P(\text{Case}_i) \times \text{runtime for Case}_i$$

Muddiest Points

- **Q: what is the meaning of life**
- I think this is outside the scope of this course
- You can build algorithms to make people's life better
 - Keep people and societal impact as a major factor in design decisions